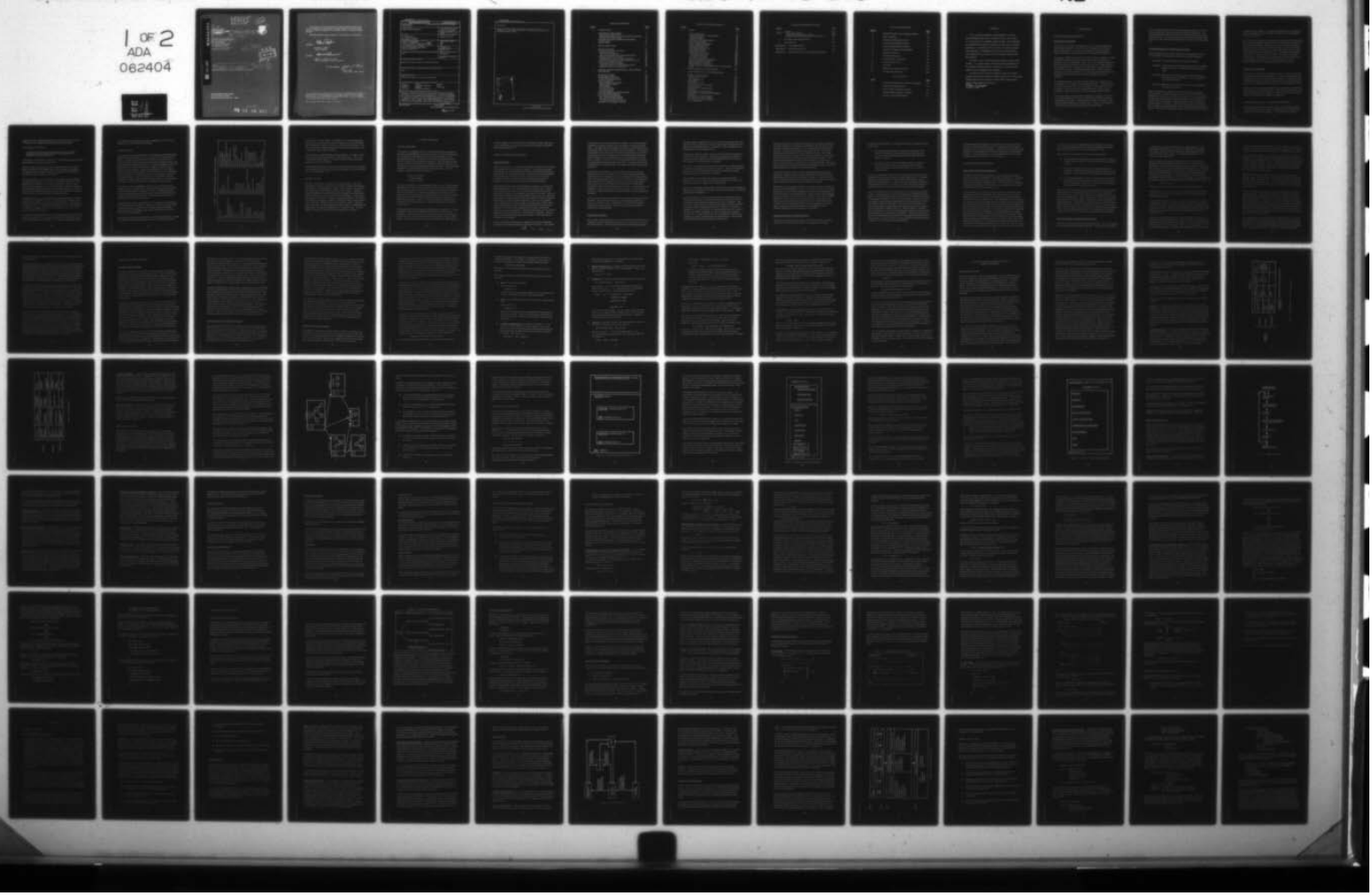
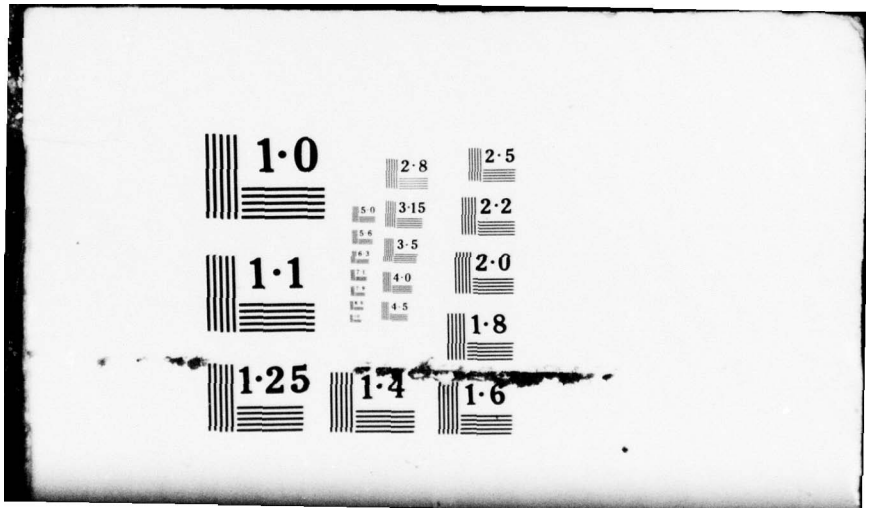
AD-A062 404 HONEYWELL INC MINNEAPOLIS MN CORPORATE COMPUTER SCIE--ETC F/G 9/2
RATIONAL DESIGN METHODOLOGY.(U)
SEP 78 D BOYD, A PIZZARELLO, S C VESTAL F30602-77-C-0043
UNCLASSIFIED 47338 RADC-TR-78-208 NL

1 OF 2
ADA
062404




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6

LEVEL II

12

(18) (19) RADC-TR-78-208
Final Technical Report, 9 Feb 77-19 May 78
Sept [redacted] 78

(12) 181 p.

(6) RATIONAL DESIGN METHODOLOGY,

(10) Dr. Donald/Boyd,
Dr. Antonio/Pizzarello
Mr. Stanley C./Vestal

Honeywell Incorporated

(14) 47338

(15) F30602-77-C-0043 (16) 5550 (17) (18)

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

410 981

78 12 18 051

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-208 has been reviewed and is approved for publication.

APPROVED: William E Rzepka

WILLIAM E. RZEPKA
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, COL, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-78-208	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: RATIONAL DESIGN METHODOLOGY	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 9 Feb 77 - 10 Mar 78
	6. PERFORMING ORG. REPORT NUMBER: 47338
7. AUTHOR(s): Dr. Donald Boyd; Dr. Antonio Pizzarello; Mr. Stanley C. Vestal	8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0043
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Honeywell Incorporated, Corporate Computer Sciences Center, SSRC, 10701 Lyndale Ave So, Minneapolis MN 55420	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 5550, 1804
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIE), Griffiss AFB NY 13441	12. REPORT DATE: September 1978
	13. NUMBER OF PAGES: 182
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: William E. Rzepka (ISIE)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

design	computer	algorithm
methodology	structured programming	proof
software	WELLMADE	correctness
abstraction	constructive approach	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes an effort to specify a software design methodology applicable to the Air Force software environment. Available methodologies and techniques were examined and investigated for (1) level of completeness; (2) ability to conform to Air Force design practices; and (3) inclusion of techniques for proof of correctness, design specification, and performance assessment of static designs. The rational methodology selected is a synthesis of ideas including data abstraction and refinement, constructive ap-

(see reverse)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

410 981

20 (Cont'd)

proach for software design, documentation procedures and tools. As a demonstration, the methodology was used to design a major function in the IBM Program Support Library.

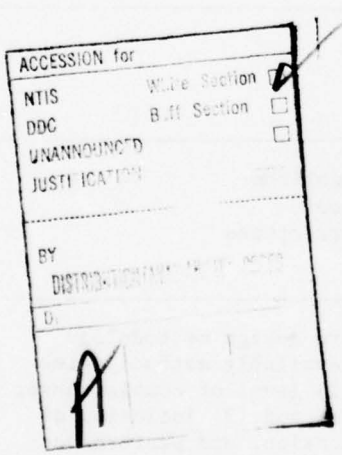


TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

EVALUATION

This final Report represents the development of a software design methodology specifically tailored for use by the Air Force in its acquisition of C3 embedded computer systems. This methodology was synthesized from the best, currently available ideas in software engineering. Its principles were identified, its design procedures were established, a comprehensive set of tools was outlined and a small demonstration on a typical Air Force software design problem was performed.

This work is part of the Software Cost Reduction program (TP05) at RADC. It is oriented toward increasing current understanding of the software engineering process so that the costs associated with system life cycle management may be reduced.

The results of this work are directed at practicing software engineers and software acquisition managers working in Air Force System Program Offices and are intended to enhance their understanding of modern software engineering techniques.

William E Rzepka

WILLIAM E. RZEPKA

1. INTRODUCTION

THE SOFTWARE DESIGN PROBLEM

Introduction to the Problem

The annual cost of software development in the U.S. is approximately 20 billion dollars [BOEH77]. In FY 1975, the U.S. Department of Defense spent an estimated 2.9-3.6 billion dollars [FISH74] yet this industry is afflicted by problems which actually slow down computer introduction.

Furthermore software products are not renown for their reliability. A large percentage of programming labor is required to repair malfunctions in products in use for long periods of time. The cost of software maintenance has been estimated as 70-75% of its total life cycle cost. Although not all the maintenance is for repairing errors, the economic impact of software errors is very sizable. It has been estimated in one aircraft computer application, software development costs were approximately $75 per instruction while during maintenance the costs were $4000 per instruction [TRAI73].

The overall impression of software industry observers is that there is a considerable lack of control of the development process. The product does not reflect the requirements, the requirements are not understood by the developers, or the time and cost schedules are overrun by an order of magnitude. Each of these show a lack of development process control.

It was fashionable in the late sixties to talk about a "software crisis" and many technical papers were written analyzing the problems and causes, and suggesting remedies. At the same time, results of computer science research supplied the basis for possible cures. Today, we can apply this scientific knowledge to produce programming methodologies.

This document presents a review of programming principles and analyzes their application to working procedures for software development. To accomplish the latter task, a definition of a software development methodology is essential. The software product life cycle will be analyzed in the framework of the traditional development environment to arrive at a definition.

Software Development in a Traditional Environment

The development cycle is divided, following the "Management Guide to Avionics Software Acquisition" [LOGI76], into the following phases:

Conceptual - during this phase requirements are analyzed.

Validation - requirements are transformed into some form of system design and therefore accepted for further development work.

Full scale development - the requirements are transformed into a functioning product (a program or a set of programs for the required hardware).

Production/deployment - during this phase the functioning product is distributed to the users.

These four phases are identifiable in all software development plans. A characteristic of software development in this mode is the difficulty of defining the transition from one phase to the next. In fact, the production of code usually shows problems which are actually due to incomplete or misunderstood requirements. These problems should be recognized in previous phases. The result is that the developers perform activities of the conceptual and validation phases during development and the production phases. Other product developments suffer from these problems but not to

the extent found in software. In software development too many efforts fall outside this sequence of events making it difficult to exercise meaningful control on the process.

This situation demonstrates an inadequate understanding of the programming task. Two major problem areas are obvious: first, the requirements are not sufficiently complete or understood; second, the actual code does not represent a faithful rendition of the required task. The second problem is usually identified as that of program correctness and has received considerable attention in the research community. The problem of ill-defined requirements is harder to formalize, since it is caused by inadequate communication. Fortunately, the work on program correctness has supplied insight into the whole cycle of program development so that a heuristic approach to the requirements problem may be outlined.

The Need for a Methodology

Improvement of the software development process must be in accord with the scientific principles of programming. The mere recognition of these principles does not improve the development process. The theory is hard to understand and there is a need for procedures that can be followed by a large variety of people assigned to all phases of the development process.

Another aspect of the successful application of principles is to support the work procedures with appropriate tools. This makes the application of the procedures easier and more natural. In other words, there is the need to define a rational software development methodology.

DEFINITION AND SCOPE OF A RATIONAL METHODLOOGY

The emphasis of this study for the Air Force is on the design problem. This does not mean that the aspects of implementation and deployment of software are considered uniportant. It is the opinion of the authors,

supported by wide ranging experiences, that the most improvement can be obtained in the design phase [BOEH75] and its methodology.

A methodology is defined as

> a complex of work procedures supported by a body of scientific principles which are made effective by an appropriate set of notational and organizational tools.

Furthermore, this definition is a basis for a methodology which may include implementation and production/deployment.

Software design includes all the activities beginning with the conceptual phase and ending with the detailed specifications of data and algorithms. The design must be validated to obtain a very high level of confidence in its abilities to satisfy problem requirements.

The key advantage of a well-considered methodology over the traditional development environment is in the level of confidence obtainable. In the traditional environment there is no way of knowing if a design is satisfactory until the programs run on the real machine. Therefore, the traditional design procedures yield a very low level of confidence. It is the rule rather than the exception that design flaws are discovered by the execution of the actual code during the production/deployment phase.

A design methodology is also required to accommodate changes in problem requirements. Frequently, a number of relatively minor changes in the problem specifications occur during the product life cycle. The usual way of coping with this is to change the executable code and test to demonstrate the adequacy of changes. The design documentation, if it exists, is frequently neglected in this process.

In a rational design methodology, problem specification changes are treated as new design problems. A complete validation and documentation of the solution is made before actual code is run. The task of designing changes

is facilitated by the documentation of the original design. This is a key element of a successful design methodology.

CONTRACT TASKS

The final contract report deliverables for the Rational Design Methodology (RDM) effort will be delivered in two versions: interim report and final report. The interim report contains the results of work performed to satisfy Statement of Work (under Goals and Purpose of Support Tools, and Design Notations categories of Section 4). The first paragraph requires the examination of current software design, validation, and performance assessment methodologies. The second paragraph calls for the identification of a promising methodology from those examined. The methodology may be a synthesis of concepts and facilities from a number of the most promising candidates. The work required to accomplish this effort is contained in Honeywell tasks "Software Design Problem," and "Definition and Scope of a Rational Methodology" (at beginning of this section).

The study of current methodologies was done in the first month of the contract period. An initial list of candidate methodologies and languages relating to methodologies is displayed in Table 1. Existing literature searches, personal contacts and interactions with RADC personnel were conducted to identify those features of a methodology which were most relevant to the Air Force environment.

During the second month of the contract, the concepts of the methodology were formulated and documented. The basis for this was the methodology called WELLMADE, currently under development within Honeywell. Specifications of WELLMADE may be found in the references [MCGE76, BOYD76, PIZZ76, HONE77a].

The remaining work, subsequent to the two initial Honeywell tasks, has been documented in monthly technical status report, and in this final report.

Table 1. Candidate Methodologies and Related Languages

WELLMADE	DESIGN	DIJKSTRA/HONEYWELL
DES	DESIGN, SPA	MULTICS/HONEYWELL
LIS	PROGRAM LANG.	CII-HB/HONEYWELL
TOPD	DESIGN	HENDERSON, SNOWDON/NEWCASTLE
STRUCTURED PROGRAMMING	DESIGN, IMPLEMENTATION	MILLS, BAKER/IBM
COMPOSITE DESIGN	DESIGN	CONSTANTINE, MYERS/IBM
JACKSON DESIGN	DESIGN	JACKSON
LOGOS	DESIGN	ROSE, et. al./CASE-WESTERN U.
NUCLEUS	PROGRAM LANG.	GOOD, RAGLUND/U. OF TEXAS
PASCAL	PROGRAM LANG.	WIRTH, HOARE
CLU	PROGRAM LANG.	LISKOV/MIT
HIERARCHICAL (SPECIAL)	SPECIFICATION	ROBINSON et. al./SRI
CONCURRENT PASCAL	PROGRAM LANG.	HANSEN
HOS	DESIGN	HAMILTON, ZELDIN/HOS INC.
ISDOS (PSL/PSA)	REQUIREMENTS, DOCUMENTATION	TEICHROEW/U. OF MICH.
SDS (SREM/RSL)	REQUIREMENTS	DAVIS, VICK/BMDATC
SDS (PDS/PDL2)	DESIGN	DAVIS, VICK/BMDATC
SADT	REQUIREMENTS/DESIGN	ROSS et. al./SOFTECH
ALPHARD	DESIGN	WULF, LONDON, SHAW
ALGEBRAIC	SPECIFICATIONS	GUTTAG, et. al.
MODULA	PROGRAM LANG.	WIRTH
EUCLID	PROGRAM LANG.	LAMPSON, et. al.
SEMANTIC	REQUIREMENTS, DESIGN	SERRINTINO, MILLS/IBM

In particular, "Contract Tasks," in Chapter 1 and "General Discussion" and "Proof-of-Correctness Principle" of Chapter 2 were concerned with the specification of concepts, procedures and general design tools. This work completed the detailed examination of the general design phase, and terminated at approximately month six.

Honeywell task 3, Detailed Design (SOW paragraph 4.1.3) began at month one and continued through month nine of the contract term. This work concentrated on PDL notation, verification tools and static performance assessment tools and techniques.

Months eight through twelve were devoted to the demonstration of the RDM on a selected portion of the IBM Programming Support Library (Honeywell task 4, SOW paragraph 4.1.4).

REPORT OUTLINE

The body of this report is arranged in six major sections. The first, Introduction, contains a discussion of the software problem and the Honeywell definition of a methodology. Chapter 1 also contains a description of the contract tasks performed and the outline of the report. Chapter 2, Design Principles, contains descriptions of the scientific principles which must form the basis for a successful design methodology. In particular, the principles of proofs-of-correctness, limiting complexity, and constructive design are expounded. Chapter 3, Design Procedures, describes the way in which the scientific principles are applied to a RDM. Chapter 4, Tools, describes aids which support the RDM. Chapter 5 demonstrates the application of the selected methodology on a software design. The sixth chapter, Conclusions, summarizes the findings of this research.

2. DESIGN PRINCIPLES

GENERAL DISCUSSION

The weakness of traditional software design techniques became apparent ten years ago [NAT069] and led to active research in software engineering. This research has highlighted the difficulties and inadequacies in the traditional approaches. Among them are improper understanding of the software requirements, inadequate techniques for verifying software satisfaction of intended requirements, the inability to cope with large and complex software, and uncontrolled design process activities. Three principles of software design have been identified, and are the center of active research. They are referred to herein as:

proof-of-correctness
limiting complexity
constructive design

"Proof-of-correctness" is a mathematically rigorous proof that the detailed specifications of software design completely carry out their requirements. This assumes the designer has correctly identified and specified the requirements of the software. By "limiting complexity," the problem is decomposed into a set of intellectually manageable steps, each requiring a limited number of design decisions and each producing design specifications of limited size and detail. Finally, "constructive design" is intended to be a constrained and controlled process of stepwise software design and proof-of-correctness.

The foundations of a rational design methodology are a synthesis of concepts resulting from the theory developed through research on these design principles. In this section, the research involving each of these principles is discussed in the context of the concepts used in software design methodologies. The procedures which lead to comprehensible and mathematically

provable designs are necessary tools for any software project since an incorrect program has no value and a program which cannot be understood has at best marginal value.

PROOF-OF-CORRECTNESS PRINCIPLE

Program Semantics

Research on the proof-of-correctness principles has usually been concerned with demonstrations of functional correctness. Assuming the functional requirements are correctly specified, the proof must demonstrate that the program satisfies its intended functions. The notion of this demonstration of correctness was mentioned as early as 1947 by John von Neumann [GOLD63]. In order to make such a demonstration, the program's *semantics* must be defined in a concise and uncompromising way.

The traditional technique for defining program semantics is known as the operational approach. This was expressed by John McCarthy in 1962 as: "The semantics itself is given by an interpreter that describes how the state vector changes as the computation progresses" [MCCA63]. Thus, the most direct technique for determining what a program does is by observing its execution. Testing a program shows functional correctness of program path for a specific set of values of input data. However, to completely verify a given program, every possible computation (path and data values) must be demonstrated. This is generally impossible, even with simple programs. The result of this impossibility was clearly identified by Dijkstra in his famous statement: "Program testing can be used to show the presences of bugs, but never to show their absence" [DIJK72] . However, testing remains the predominantly used technique for demonstrating the correctness of a program.

It was not until 1967 that Floyd first suggested an execution-independent technique for defining the semantics of programs [FLOY67] . This method

is often known as the inductive assertion technique. The functional specifications of a program are given by a description of its input and output states. The state space of the program is defined as the collection of all the possible values of all the variables used by the program. The input and output states are subspaces which are defined by assertions which further constrain the values of variables and describe relationships existing between variables at start and completion time in the program's execution. Thus, a program is correct if it can be demonstrated that when it begins in an input state satisfying the input assertions, it traverses several intermediate states and eventually halts in an output state satisfying the output assertions.

To make this demonstration, it was necessary for Floyd to define the meaning of each command construct as a condition relating antecedents (input assertions) to consequences (output assertions). By using only flow charts in which meanings of constructs are defined and by inserting assertions between the flow chart blocks characterizing intermediate states, Floyd could make a clear demonstration that the correct relationship (verification condition) exists between two assertions. In particular, it was possible to verify correctness of an entire program. Floyd limited the constructs to sequential commands containing boolean tests and assignment statements.

In 1969, Hoare formalized this technique for high level language constructs [HOAR69]. Hoare defined the semantics of the language constructs by axioms and inference rules. Consequently, the execution independent technique for defining program semantics and verifying functionality is called the axiomatic approach.

The Axiomatic Approach

The concepts of Floyd and Hoare are summarized here because they are the foundations for all techniques involving proof-of-correctness. Let S be the text of the program. Let P and Q be assertions characterizing the input

and output states, respectively, of S. To verify S with respect to its antecedent P and its consequent Q, it must be proved that if P holds before execution of S, then Q will hold after execution of S. This verification is denoted by Hoare with the notation P{S}Q.

The program constructs whose semantics were first defined by Hoare are compositions of the operations: assignment, alternation, and iteration. It has been shown by Bohm and Jacopini [BOHM66] that these constructs are sufficient for representing any computation.

The rule for composition of operations allows proofs of a sequence of statements to be decomposed into a sequence of proofs. That is, P{S1;S2}Q may be proven by first proving P{S1}R for some intermediate set of states characterized by R, and then proving R{S2}Q.

The consequence of an assignment statement is given by the axiom: $P[x \rightarrow e]\{x := e\}P$, where $P[x \rightarrow e]$ is the predicate obtained from P when every free occurrence of x is replaced by the expression e.

The rule for alternation requires that each alternative be proven separately. Thus to prove P{if B then S1 else S2}Q, it must be proven that (P and B{S1}Q) and (P and not B{S2}Q) hold.

To provide rules for iteration, Hoare introduced an invariance theorem. An invariant assertion is a relationship between data structures which is true each time control enters a specific point in the computation. Thus the proof of a loop, P{while B do S}Q requires that an invariant assertion I be found and that the following properties be verified: the invariant is implied by P, P I, the invariant is maintained throughout each iteration of the loop, I and B{S}I, and finally, the variant implies the consequent, I and not B Q. To show that the invariant is maintained throughout loop iteration, an inductive argument must be used [REYN76]. The invariant relationship describes the functionality of the iteration.

Note that in the above discussion involving proofs of iterations, nothing was said about whether or not the iteration terminates; thus, the proof is valid only when the program terminates. This is called a proof of partial correctness. To show that the program terminates correctly for all states satisfying the input assertions, termed total correctness, a decreasing positive valued integer function which is bounded from below must be found. With each iteration, it must be shown that this function decreases. The iteration must eventually terminate since the function is bounded from below. These functions are characterized by well-founded sets. The use of these sets for proving termination has been investigated by Floyd [FLOY67], Manna [MANN69], and Manna and Pnueli [MANN74b].

An obviously missing construct was the unconstrained branch (the go to statement). Although initially controversial [SIGP72, KNUT74], the elimination has been accepted by most program designers. Researchers continue to define the semantics of new control structure constructs to eliminate these controversies [CLIN70, ZAHN74, ADAM75, KOWA77].

Illustrations of the axiomatic approach to proving correctness are found in the text by Manna [MANN74a]. Properties of induction and its use in variations of the Floyd and Hoare techniques are discussed by Reynolds and Yeh [REYN76] and Basu and Misra [BASU75]. These variations include subgoal induction [MORR77], structural induction [BURS69], intermittent assertions [MANN76], and weak logic of programs [LUCH77]. Properties of side effects have been investigated by Clint [CLIN73], Hoare [HOAR71], Ernst ERNS77], and Kowaltowski [KOWA77]. Synchronization constructs and their semantics are being investigated by Hansen [BRIN73], Hoare [HOAR74, HOAR72c], and Owicki and Gries [OWIC76].

Significant Concepts and Additional Research

The axiomatic approach as described by Floyd and refined by Hoare and later research efforts remains the primary technique for providing

correctness proofs [ELSP77]. The significant concepts resulting from this research are:

1. The functional specifications of a program are given by static (execution independent) representations of its state space and through assertions characterizing the relationships which hold between its objects.

2. The notation used to describe a program design is restricted to those constructs which are semantically defined by their input-output state transformations in the form of axioms and inference rules.

Much of the methodology research involves searches for notations for representing functional specifications [PARN72a, NOON75] and for representing program designs [CHU76, LOND70, GRIE77, DIJK76] by constructs with well-defined semantics. In cases where the methodology supports the actual demonstration of a correctness proof, the underlying assumption is that the proof will be carried out after the program has been designed.

It is a very complex task to formally prove the correctness of an arbitrary program. These proofs have only been presented for relatively small programs. Isolated cases have been carried out as "hand-simulated" proofs of large programs [LOND70]. There is a large amount of research on correctness proofs aimed at machine implementation for semi-automatic verification [KING71, GOOD75, WALD73]. This research involves systems for semi-automatically constructing invariant assertions derived from the program text [WEGB74, GERM75, KATZ77, TAMI76, DERS77]. Extensive syntax checking of the assertion and specification language is used to reduce the complexity of the verification task [ROBI77]. Finally, Dershowitz and Manna are investigating the possibility of translating invariant assertions to new assertions in order to perform program modifications adapting program models to fit various program designs [DERS76].

Any methodology which supports in a practical way the concepts derived from the correctness proof principle must support concepts for limiting complexity and for constructive design. It is necessary to decompose large programs into manageable small programs through some modularization technique. It is also necessary to use knowledge obtained from the proof technique to guide the design of a correct program.

LIMITING COMPLEXITY PRINCIPLE

Abstraction as a Basis for Decomposition

The highly complex task of designing a correct program has motivated research on effective techniques for reducing this complexity. The goal of this research has been to find ways to decompose the design task into a series of intellectually manageable steps. The concept which is central to decomposition techniques is the effective use of abstraction to eliminate unnecessary detail from the design process.

In 1968, Dijkstra successfully demonstrated the use of abstraction to decompose the design of an operating system into a hierarchy of modules [DIJK68]. Each level in the hierarchy represented an abstract concept which could be used at a higher level. Thus the highest level represented a very powerful machine with highly abstract operations which were realized at lower levels. By constraining the interface between levels, Dijkstra was able to show that the correct behavior of the system was dependent only on the correct behavior at each level. Furthermore, each level of abstraction was concise enough to be implemented and tested before the next higher level was built and tested. Dijkstra later described an example where the abstract levels were identified in a top-down fashion by designing a highly abstract program and identifying its lower level abstractions as part of the design process [DIJK72]. The modularization was hierarchical with each abstract level representing a refinement of an upper level. This successful use of abstraction as part of the design process was also achieved by other

researchers such as Naur [NAUR69] and Wirth [WIRT71], each using a form of top-down design to identify the abstract concepts at the highest levels and refinement of these concepts at the lower levels.

Three important concepts have been taken from this research:

- The decomposition of programs into modules should be based upon carefully selected abstract concepts representing a single design decision.

- The use of an abstract object requires only knowledge of its semantics. Thus the specification of an abstract concept is independent of its data representation and algorithm implementation.

- The relationship between modules is hierarchically ordered permitting top-down design techniques to be used to constructively identify the abstract concepts and to constrain these interfaces to demonstrably correct program behavior.

Thus, the use of abstraction in the design process has become the modularization technique for providing the program structures. Constantine [STEV74] has given a set of module attributes for evaluating the effectiveness of the modularization. These attributes are given in terms of module cohesiveness and module coupling. Cohesiveness refers to the degree of intra-module unity. A high degree of cohesion within a module is accomplished by equating a module to a single function or to a set of related functions. Module coupling refers to the degree of connections between modules. A low level of coupling is desirable and is accomplished by reducing the number of objects directly referenced in one module by another module.

Abstract Operations and Abstract Data Structures

Two forms of abstraction are used in program design. The more traditional form of abstraction is the abstract operation. This form has a high degree

of cohesiveness as it often represents only a single operation. This is implemented by subroutines or procedures in a programming language. The designer need only know the input-output specifications to use the operation at high levels of design. The detailed implementation may be deferred to later design activity.

A recent approach to program decomposition similar to that used by Dijkstra is through abstraction of data structures. A data structure is identified by a collection of abstract operations which access or modify the data values and an assertion defining the relationships which must be maintained among the components of the structure. This assertion is called the data invariant [HOAR72a] and is similar to the one used in the semantic definition of the loop construct. A simple example of a data invariant is the condition that the domain size of an array be positive. This invariant relationship is between the high index (hib) and low index (lob) of the array, and is

$$hib - lob + 1 \geq 0.$$

Thus all array operations must maintain this invariant relationship.

This approach is made more precise by the introduction of the concept of data type. This concept is a fundamental one in the design methodology WELLMADE [BOYD78].

It is intuitively clear that any datum is used to encode a piece of reality by using the datum values. It is also necessary to identify a name which is used to refer to a set of values, all encoding different manifestations of the same piece of reality. In this view all data are pairs (name, value) having some specific significance with respect to the problem the program is called to solve.

The range of values of data defines a state space which can be used for encoding different, although similar, pieces of reality. For example, the state space formed by definition of an integer array can be used (by giving distinct names) to represent a pointer table or an n-dimensional vector or any other suitable entity. It appears convenient to identify the common set

of values by naming it separately from the specific data, as in the case of the above example for the set of values named as "integer array." The set of possible values that a datum may assume is called the type of the datum.

The set of values, however, is not sufficient to fully characterize fully a data type. For example, a date is not distinguishable from any integer used to indicate number of dollars, in the sense that the range of values is in both cases a subrange of integers. However, while the operation of adding two numbers representing dollars makes sense, the same operation is senseless for two dates, while their subtraction would be perfectly legitimate (to create a value of the type "age").

Therefore the complete characterization of a data type is obtaining by defining a range of possible values (a state space) together with the operations permitted in that state space. Certain objects of the real world are considered elementary, others are seen as association of more elementary ones. This fact requires that data types can be composed to create appropriate structures.

The idea of abstraction of data structures can be formalized by defining abstract machines. To describe the semantics of an abstract operation, appropriate relationships between variables in the state space of the program must be defined. In general it is necessary only to put in evidence those properties which are necessary to define the semantics of the operation as seen at that level of definition. This is accomplished by identifying abstract data types in such a way that the static definition of the abstract operation is not unduly complicated. This second form of abstraction is much more general and permits the design of systems of programs operating against a permanent data base.

Since a program at every level may need the definition of one or more abstract operations, the complete definition of their semantics becomes the definition of a new state space (the collection of data values) that together with the abstract operations (the only legitimate ones on the abstract data) form a new data type in an abstract machine. The program using the

abstract operations can thus be proven in the environment supplied by the abstract machine.

Specification techniques for abstract data structures have been proposed by Parnas [PARN72b], Guttag [GUTT77], and Zilles [ZILL76]. WELLMADE specifications of abstract data structures include a specification of each operation and the invariant relation on the state space which must be maintained by these operations. Simple examples of data invariants for abstract data structures are the successor/predecessor relationship nodes of a graph, or the lexical order of elements in an array. The key to identifying abstract data structures is often through the recognition of the data invariant.

Parnas observed that abstract data structure operations were either value producing or cause a change in the state of the data structure. Thus each operation was classified as either a V-function (value), O-function (state transformation), or OV-function. The specification of the module involves a description of the effects of each of its functions. Modules whose operations can be specified in this fashion are referred to as Parnas modules. SRI [ELSP77] and MITRE [MILL76] have made affective use of the Parnas modules for specifying abstract machine hierarchy capabilities. Each abstract machine is a collection of Parnas modules. This work has been restricted mostly to specifications of operating system designs.

Several modern programming languages support the implementation of abstract data structures, [DAHL72, LISK74, WIRT77, WULF76, ICHB74]. Most of these implementations are based upon the class concept first introduced in the language SIMULA [DAHL68]. The class represents a permanent data structure accessed only through its operations. A variation of the class is a monitor which permits access to its data structure through activation of its operations by concurrent processes [BRIN73, HOAR74]. Thus, synchronization rules are implied in the implementation to provide the proper synchronization of these processes. This concept has been used successfully by Hansen [BRIN76] for operating system design.

CONSTRUCTIVE DESIGN PRINCIPLE

Correctness Based Guidelines

It has been emphasized at several places in this report that correctness considerations must be part of the design process. Thus a rational design methodology must include features which allow the designer to use the functional requirements as an aid to deriving a correct program design. The techniques described in the previous section are aimed at making the design process more manageable by using abstract concepts to reduce the complexity of the problem. However, these methods do not suggest the techniques for solving the problem. That is, a design must be discovered for representing a program which begins in a state satisfying input assertions and halts in a state satisfying output assertions. This representation is the sequence of operations which must be performed to make the state transformation.

The problem of constructive design is to discover a discipline which will permit the designer to begin with a set of requirements for a program module and then to systematically construct a software design to satisfy the requirements. The foundations of most current design disciplines are taken from the concepts and techniques used to demonstrate a proof-of-correctness and to decompose a program. These concepts and results have been used to describe a set of procedures and "good-practices" to be used as part of the design process. In most of these disciplines, the software designer must perform a correctness proof after the design is completed. The design discipline has not provided for a technique of validating the correctness of each step of the design before proceeding to the next step.

There are two parts of a design discipline involving the decomposition of the design process into a series of manageable design steps. The first part involves correctness preserving design steps, while the second involves use of the requirements. Guidelines for decomposition are presented as a set of rules which must not be violated in order to preserve the capability to

demonstrate a correctness proof. These rules are generally derived from the rules of inference supplied by Hoare and Floyd for defining the semantics of control constructs, and rules for partitioning the data space to preserve the invariance of data relationships [MILL75]. Most of the existing design methodologies use an informal application of these rules as a design discipline. Modularization and design guidelines are based upon correctness-preserving rules, but very little is said about how the program requirements are used to guide the design process. That is, they are not constructive. These methodologies suggest a top-down method such as a stepwise refinement, or levels of abstraction for general design, but do not supply a discipline for deriving the algorithm of the module at any level.

Among the methodologies using this type of discipline are Structured Design [MYER75], HOS [HAMI76], SRI [ELSP77], and MITRE [MILL76]. Each of these methodologies defines a set of guidelines for module selection. The guidelines of Structured Design are based upon the cohesiveness and coupling attributes described by Constantine. SRI and MITRE use a form of Parnas module for specifying abstract data structures. The guidelines for module selection used in HOS are derived from a set of six axioms concerned with the control of the interfaces between modules. The axioms are reported to be necessary and sufficient for rigorous demonstration of module interfaces consistency and thus act as the guidelines for module selection. Both Structured Design and HOS use functional abstraction rather than data structure abstraction as the basis for module decomposition.

Guidelines Based on Functional Requirement

Two methodologies which have more constructively oriented design disciplines are Jackson's technique [JACK75] and the Warnier Method [WARN]. They assume that the major function of a program is realized by a loop and may be described by an invariant relationship on its variables which can be established at the beginning and termination of the loop. The requirements of the program may be described informally in terms of the invariant relationship and thereby act as a guide to the design process.

The only design discipline which is constructive is the method of predicate transformers defined by Dijkstra [DIJK75]. This design discipline is used in the Honeywell design methodology WELLMADE [BOYD76, PIZZ77]. In this approach a set of rules are given to the designer for deriving the program design from the functional requirements. That is, the design activity begins when the designer has given the specifications of a program's input states, output states, and data invariants. Using only these specifications, the designer applies a sequence of derivation rules to the specifications resulting in a sequence of design constructs. These rules form the semantics of the design constructs. Thus when the rules are properly applied the notation (involving the design constructs) specifies a correct design. In this way the derivation of a correctness proof and design specification proceed in parallel. This notation is used by WELLMADE and referred to as p-notation. Wegbreit has recently suggested that specifications and justifications be constructed in parallel with the code and be included as part of the program text [WEGB77].

The concepts described by Dijkstra form a calculus for program derivation. The semantics of the program constructs are the derivation rules. Semantics of a construct characterize the largest possible set of input states for which the execution of the construct will halt yielding a desired output state. Thus, if the desired input state is a subset of the derived input state, the construct has been correctly chosen. Note that this design procedure generally starts with the resulting output state and then proceeds backwards searching for constructs which result in the output state and characterize the desired input state.

Dijkstra's Constructive Approach

Since Dijkstra's approach to program design is central to a rational design methodology, the major components of this technique are now briefly summarized. The program design constructs proposed by Dijkstra are referred to as guarded commands. These form a language sufficient for describing the design of a program's detailed logic. There are six constructs

corresponding to commands for no-operation, an abort operation, sequence of operations, replacement operation, alternation, and iteration. The two guarded commands are derived from sets of statements, found in the alternation and iteration constructs, preceded by a guarding boolean expression. That is, the statement lists are eligible for execution only if the boolean expression is true. Furthermore, since both alternation and iteration allow multiple guarded statement lists and since no order for evaluating the guards is implicit in the language, a nondeterministic program may be represented by the language.

To define the semantics of the guarded commands, Dijkstra introduced a predicate transformer function, wp, which acts upon a system of predicates and mechanisms. Let P be a predicate characterizing the output states for some program test S. P is called the postcondition and S denotes an underlying mechanism. wp(S, P) characterizes the initial states of the mechanism S such that activation of S in any state satisfying wp(S, P) will certainly lead to a properly terminating activity leaving the mechanism S in a state satisfying the postcondition P. wp(S, P) is called the weakest precondition of S which satisfies P. After defining a set of properties of wp, Dijkstra defined the semantics of a set of mechanisms (the guarded commands) by their state transformational properties by describing the corresponding predicate transformational properties.

The major difference between Hoare's axiomatic system and Dijkstra's axiomatic system for defining semantics is the emphasis placed on using the constructs for building a correct program rather than demonstrating that a program has been built correctly. The construction of a correct program must provide for its termination. Dijkstra has characterized the necessary and sufficient preconditions for a mechanism to terminate satisfying the desired postconditions whereas Hoare characterized only sufficient preconditions for a mechanism to satisfy the postcondition if it terminates. By combining Hoare's and Dijkstra's notation, this can be displayed as:

$$\text{if } Q \Rightarrow wp(S, P) \text{ then } Q\{S\}P \text{ and } S \text{ terminates}$$

where Q characterizes the input states specifications of the program.

The program designer must construct a program which maintains the relationship $Q \Rightarrow wp(S, P)$. Conversely, if a program is constructed and proved to be partially correct then the following relationship holds:

$$\text{if } Q\{S\}P \text{ then } wp(S, P) \Rightarrow Q$$

That is, Q is not a necessary precondition since the program may not terminate.

The syntax and semantics of the six guarded commands are now briefly discussed.

- Skip is a no operation command

  The semantics are:

  $$wp(\text{skip}, P) = P$$

  Thus the "skip" causes no state change to occur and therefore the weakest precondition is the same as the postcondition.

- Abort is an operation which leaves the system in an undetermined state.

  The semantics are:

  $$wp(\text{abort}, P) = F$$

  where F is the constant predicate denoting the condition which is satisfied by no states. That is, the weakest precondition for an abort which satisfies P characterizes no states in the underlying state space.

- A sequence of operations is denoted by a semicolon, ";". The semantics describe the way in which a program may be constructed in reverse by starting with the desired postcondition and then constructing commands in the reverse order in which they are applied. This is given by

  $$wp(S_1;S_2, P) = wp(S_1, wp(S_2, P)).$$

Note that this reverse order of construction may continue until the relationship $Q \Rightarrow wp(S, P)$ is satisfied.

- Replacement operations are defined with precisely the same predicate as indicated by Hoare's axiom of assignment. That is, the semantics are:

  $$wp(x:=e, P) = P[x \rightarrow e].$$

- Alternation is a guarded command whose general form is given by:

  $$\underline{\text{if}}\ B_1 \rightarrow SL_1 \ \square\ B_2 \rightarrow SL_2 \ \square \ldots \square\ B_n \rightarrow SL_n\ \underline{\text{fi}}$$

  where $B_1, B_2, \ldots, B_n$ are boolean expressions representing guards and $SL_1, SL_2, \ldots, SL_n$ representing the corresponding guarded statement lists. The semantics are given as:

  $$\begin{aligned} wp(\underline{\text{if}} \ldots \underline{\text{fi}}, P) = {} & (B_1 \ \underline{\text{or}}\ B_2 \ \underline{\text{or}} \ldots \ \underline{\text{or}}\ B_n)\ \underline{\text{and}} \\ & (B_1 \Rightarrow wp(SL_1, P))\ \underline{\text{and}} \\ & B_2 \Rightarrow wp(SL_2, P))\ \underline{\text{and}} \\ & \ldots\ldots\ldots\ldots \\ & (B_n \Rightarrow wp(SL_n, P)) \end{aligned}$$

  The first term $(B_1 \ \underline{\text{or}}\ B_2 \ \underline{\text{or}} \ldots \ \underline{\text{or}}\ B_n)$ requires that at least one guard be true, otherwise the alternation aborts. The remaining terms require that each guarded statement which is eligible for execution, lead to the desired state.

- Iteration is a guarded command whose general form is similar to that of the alternation construct and is given by:

  $$\underline{\text{do}}\ B_1 \rightarrow SL_1 \ \square\ B_2 \rightarrow SL_2 \ \square \ldots \square\ B_n \rightarrow SL_n\ \underline{\text{od}}$$

  As before, $B_1, B_2, \ldots, B_n$ are the guards and $SL_1, SL_2, \ldots, SL_n$ are the guarded statement lists. The semantics of iteration are defined as:

  $$wp(\underline{\text{do}} \ldots \underline{\text{od}}, P) = \exists k: H_k(P)$$

where $H_0(P) = P$ and not $(B_1$ or B_2 or ... or $B_n)$

and for $K>0$

$$H_k(R) = wp(\underline{if} \ldots \underline{fi}, H_{k-1}(P)) \ \underline{or} \ H_0(P)$$

The intuitive definition of $H_k(P)$ is that it is the weakest precondition which will terminate in the desired state after at most k selections of the guarded list. Thus $H_o(P)$ indicates that the mechanism will not be activated if no guards are true, but will satisfy the postcondition P. In this case, the mechanism behaves as a skip.

Note that in the case of the if...fi and do...od construct the semantic definition says nothing about the order of selection of the guarded command list to be executed in the case that more than one guard is true. This means that the constructs are intrinsically non-deterministic, that is, the output state is not uniquely defined by the input state but rather a full set of allowable output states is determined by one state in input.

The use of non-determinism may appear as a useless complication at first observation. However, the practical use of the guarded commands has shown that program design is often significantly simplified. An example is shown in the section on "Practical Suggestions for Static Design," chapter 3.

The definition of the weakest precondition for iteration does not appear of immediate use in practice. A more useful result is the loop invariant theorem which is easily proven from the wp(do...od, R) given above [DIJK76]. This theorem presents the iteration rule in the following form:

$$(P \text{ and } wp(\underline{do}\ldots\underline{od}, T)) \Rightarrow wp(\underline{do}\ldots\underline{od}, P \text{ and non } BB)$$

where $BB = \exists\, i:(1 \leq i \leq n) : Bi$. The predicate wp(do...od, T) defines all the input states for which the construct will terminate in some final state. By writing the required postcondition R as a predicate implied by P and non

BB, and by considering constructs whose termination has been proved, the iteration rule can be written in terms of predicate transformers:

$$P \Rightarrow wp(\underline{do}\ldots\underline{od}, P \text{ } \underline{and} \text{ } \underline{non} \text{ } BB) \Rightarrow wp(\underline{do}\ldots\underline{od}, R)$$

The loop invariant theorem says that in order to design a loop that terminates in a final state respecting the required postcondition R, the postcondition R must be split into a conjunction of two predicates: one is the negation of the conjunction all B's (represented by non BB); the other is a precondition of the whole loop.

Furthermore, because the predicate P also represents a necessary precondition for execution of any segment Si, P must be verified at the beginning and end of every iteration. (It could be violated at some intermediate stage during the execution of a segment, however.)

P is called the invariant relation of the loop. The role of the invariant relation can be stated informally as: starting in a state satisfying the invariant, repeat the selection and execution of some segment whose guard is true while the predicate BB is true, making steps toward the denial of BB, each and every step ending in a state respecting the invariant.

To complete the coverage of the iteration construct, some further discussion of termination is required. The necessary condition for termination of a loop, as represented by a do...od construct, is that the input state satisfies at least:

$$wp(\underline{do}\ldots\underline{od}, T)$$

However, it is in general very difficult (perhaps impossible) to determine the above predicate. Therefore a more practical expression of the condition for termination must be used.

Imagine it is possible to define an integer-valued function, C, as a function of some of the variables of the program. Also assume $C > 0$ when BB is true (and P is verified), that is, if BB is true at loop initiation the number of iterations will be greater than zero. Then require that $C \geq 0$ when BB

becomes false at loop termination. What must be proven is that at each iteration the truth of Bi will cause the execution of Si which must cause a strict reduction of the value of C. Because C is a function of the state at that iteration, the program segment Si must act in such a way that $C' < C$ is the value of the function after execution. This reduction of the value of C can be expressed by a predicate which may be interpreted as the precondition of Si required to insure termination. Let us define this predicate as:

$$wdec(Si, C)$$

which expresses the weakest precondition that must be satisfied in order to have the integer-valued function C decreased by at least one by the execution of Si. The loop invariant theorem now becomes:

$$P \text{ and } Bi \Rightarrow wp(Si, P) \text{ and } wdec(Si, C)$$

Note that the loop invariant theorem as derived above differs from Hoare's invariance theorem by the introduction of the concept of termination proof by the use of the predicate wdec(Si, C).

In many cases the semantics of the construct along with a proper specification of the output state provide sufficient information for selecting the *proper constructs.* In the case of operations involving iteration, the construction of an invariant relationship from the output state is the key to selecting the iteration. The choice of an invariant relationship and the selection of an appropriate variant function to guarantee termination are discussed and illustrated in the text by Dijkstra [DIJK76]. This process of characterizing output states and selecting program constructs involves difficult design decisions and must be learned by the designer. Papers by Gries [GRIE76], Pizzarello [PIZZ77], and Boyd [BOYD76, BOYD78] provide additional examples demonstrating this technique.

The WELLMADE methodology combines Dijkstra's constructive approach to detailed logic design with the use of abstract data structures in a top-down approach to general design. Together they form a complete constructive discipline to correct software design. WELLMADE is a basis for a rational design methodology.

3. RATIONAL DESIGN METHODOLOGY (RDM) DESIGN PROCEDURES

INTRODUCTION OF RDM

A RDM must provide, in the form of guidelines, a set of mental procedures and a work organization which guide the design process from conception of the system problem to implementation of the software design. These procedures represent the overall approach of the methodology to software design. It is the lack of enforceable procedures which has produced the current chaotic state of software development. If any procedures exist, they are generally non-technical in the sense that they are not supported by principles.

The typical system design procedures are defined as a set of milestones identifying specific periods in the design process in which certain activities must be completed. Typical milestones for software development have been identified in the software life cycle model [LOGI76, REIF75]. Unfortunately there has been no prescribed method for performing the activities leading to these milestones and it has not been established that the definitions of these milestones are appropriate. Typical examples of this situation are: testing a software implementation for verification before any satisfactory demonstrating proofs of the design correctness has been achieved; and, development of detailed designs and implementations before system requirements are understood and agreed upon.

The research, which began approximately ten years ago, on software engineering principles has led to a set of concepts resulting in several software design methodologies. They vary widely in the phases of software development to which they apply and the amount of detail included in methods and procedures. In many cases, the typical software development methodology is a very general set of procedures supported by a set of good-practice techniques and tools. Often these methodologies are identified by their techniques and tools rather than their principles. This usually leads

to only partially satisfactory results of using the methodology, although better results than if no methodology has been used.

Only a few of the current software methodologies attempt to include the complete software development cycle from requirements analysis to testing and implementation of the software. The most popular of these methodologies is the top-down Structured Programming technique proposed by Harlan Mills [MILL71, MILL73a, MILL73b, MILL74]. This methodology was synthesized by Mills [MILL72a] from the early research work of Dijkstra and others. It was first demonstrated by Mills and Baker in the development of the New York Times information system [BAKE72]. The popularity of this methodology has resulted in the most well-documented procedures to date [RADC].

Applications of this methodology have yielded mixed results. As mentioned earlier, the applications often seem to make use of the tools and techniques rather than the procedures. The newness of the methodology resulted in procedures and tools based on concepts not totally understood at the time of its development. Most methodologies following the top down structured programming approach are essentially adaptations of this approach with attempts to correct certain deficiencies or to add new techniques and tools. Examples of these later methodologies have been proposed by Liskov [LISK73, LISK72], Jackson [JACK75], Myers [MYER75], Warnier [WARN], and Tausworth [TAUS77]. These methodologies generally are oriented toward procedures for design and implementation of software under the assumption that requirements are well understood. Methodologies which include requirements analysis began with the Logos project [GLAS72, BRAD72, ROSE72] and include ISDOS [TEIC77], Softech [IRVI77], and the Software Development System being developed for BMDATC [DAVI77, ALFO76, BELL77]. Finally, the current trend is to place emphasis on more specifications, structure, and procedures which lead to proof of correctness. These methodologies are being investigated by Honeywell [BOYD77], Hamilton and Zeldon [HAMI76], MITRE [MILL76] and SRI [ROUB77].

From this survey of the existing methodologies and on the basis of the principles discussed in the preceding chapter, it is possible to give an overall definition of the RDM.

To place RDM in the proper perspective, the design problems are presented in the diagram in figure 1. In this diagram, references to existing methodologies and to sections of Chapter 2 of this report are presented to indicate existing techniques. The references in the diagram are not intended to represent a complete list of references.

The diagram in figure 1 and figure 2 present four design principles which may be applied to the solution of the three major categories of design problems, namely the design of sequential programs, the design of data base systems (machines) and the design of concurrent programs.

For each column of the diagram in figure 2, the techniques used in RDM are listed.

Structure -- The imposition of structure by means of the use of only a limited set of rigidly defined constructs is the key principle of Structured Programming. RDM will impose structure by use of the notation derived from Dijkstra guarded commands (see "Dijkstra's Constructive Approach" in chapter 2) and by use of the concept of hierarchical decomposition both by procedure abstraction and data abstraction (see abstract operations and abstract data structures, chapter 2). RDM's documentation structure is based on these concepts and is presented in the following section of this chapter.

Static Specifications -- The concept of static specifications of what a program is supposed to do is the indispensable basis for all proof of correctness (see "The axiomatic approach," and "Abstract operations and abstract data structures" of chapter 2). RDM uses the techniques described in general in the previous chapter and procedures for their application are presented in "Steps of the Design Process," "Work Procedures," and "Practical Suggestions for Static Design" sections of this chapter.

PRINCIPLES

DESIGN PROBLEMS	STRUCTURE	STATIC SPECIFICATIONS	CONSTRUCTIVE METHODS *	PROOF
PROGRAMS	MILL72a	2.2 2.4.2 FLOY67	2.4 2.4.3 DIJK76	2.2.1 FLOY67 HOAR68
MACHINES	DIJK72 2.3.2 2.4.3	ELSP77 2.4	2.4 2.4.3	HOAR72a WULF76
CONCURRENCY	BRIN73 HOAR74 2.3.2	PIZZ76 2.3.2		

* CONSTRUCTIVE METHODS ARE DERIVED FROM PROOF TECHNIQUES

Figure 1. Design Principles Versus Design Problems

	STRUCTURE	STATIC SPECIFICATIONS	CONSTRUCTIVE METHODS	PROOF
PROGRAMS	STRUCTURED CONSTRUCTS AND PROCEDURAL ABSTRACTION	STATE SPACE MODEL INPUT/OUTPUT SPECIFICATIONS	DIJKSTRA'S CALCULUS STEPWISE REFINEMENT	INDUCTIVE PROOFS FLOYD/HOARE DIJKSTRA
MACHINES	ABSTRACT DATA STRUCTURES AND PERMANENT DATA	TYPE REQUIREMENTS AND TYPE INVARIANTS DATA INVARIANTS	HIERARCHICAL TYPES CONSTRUCTION	DATA REPRESENTATION (MAPPING) HOARE ALPHARD
CONCURRENCY	MONITORS (SHARED DATA)	SYNCHRONIZATION INVARIANTS	RESEARCH	RESEARCH

Figure 2. RMD Techniques

Constructive Methods -- The principle of constructive design introduced in "Constructive Design Principle" of chapter 2 is the core of RDM. In "The Constructive Approach" section of this chapter the procedures for the use of the constructive approach are presented with examples. The use of the constructive approach together with the idea of abstract machines permits the correct step-by-step construction of complex systems. RDM procedures for the hierarchical definition of levels of abstract machines are presented in this chapter, sections "Steps of the Design Process" and "Work Procedures."

Note that in figure 2 the use of constructive design for concurrent programming is indicated as needing additional research.

Proof of Correctness -- The role of proof of correctness principles is twofold. It is the basis for the constructive design and it may be required to rigorously verify the correctness of programs and data representation designed in accord to constructive methods. In RDM the use of a-posteriori proofs is considered (whenever the reliability requirements are such to justify the additional cost) as a result of internal reviews (see chapter 3 on "Work Procedures," number 6).

DESIGN DOCUMENTATION

The result of the design process must be manifested in an appropriate documentation. This documentation will serve the purposes of specifying the implementation, of checking the correctness of design and of tracking and planning the project. There is the possibility of using an instructured documentation and undisciplined (ad-hoc defined) notation to present the result process. However, this could hardly be considered as an acceptable methodology. Therefore a documentation structure augmented by appropriate notation is necessary to furnish the receptacles for the output of the design activity.

The documentation structure can be considered from two points of view: the point of view of the writer and that of the reader. For the first point of view the structure should be conceived in such a way that the writing process is helped by supplying implicit guidelines. For the point of view of the reader the key factor is readability for checking the correctness process or for deriving the necessary information for the implementation and project planning and tracking.

These two points of view have different requirements. The order more natural to the writer is not necessarily the most natural for the reader. The importance of the checking process and, more importantly, the recognition of the need for changes in the software products, suggest that the point of view of the reader has to be emphasized. The proposed documentation scheme is based on the decomposition in abstract machines which is suggested by the limiting complexity principle.

The documentation is organized by abstract machines, in the sense that all the information relative to an abstract machine is grouped together, including the programs designed to run on the machine. The hierarchical structure is shown in the documentation structure both for the machines and the programs in each machine (see figure 3).

In the figure, each machine shown has been made up by three parts: the data types, data and programs. Furthermore, arrows indicating that data types of, say, machine 1 are realized by machine 2, 3, 4 are shown to indicate the hierarchy of abstract machines.

For each machine it is possible (and often desirable) to use procedural abstraction to design a program. For example, two of the three programs of machine 3 use another program as subroutines.

The hierarchy extends down to machines where data types are not defined by the designer but are considered primitive either because the object hardware (or language) supplies those data types or because their representation on the object hardware is standardized in some form. This is shown in the

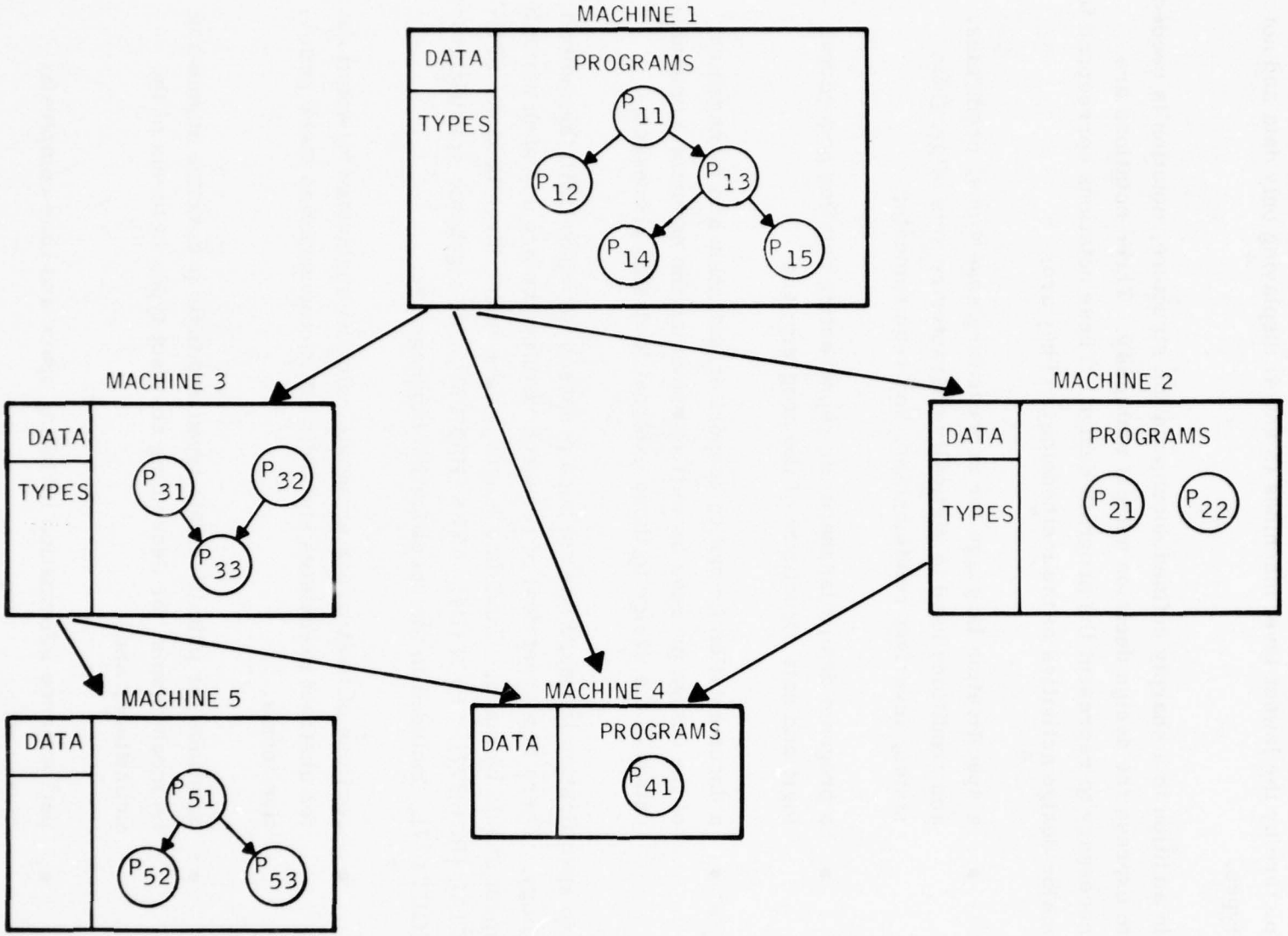


Figure 3. Program and Machine Structure

picture by the lowest level machines (5 and 4) displaying only data and not types.

In addition to a sharply defined documentation structure, notation is needed to express the design decision without ambiguity. Three notations are necessary to represent the program design. These notations correspond to major design activities of the methodology. They are:

- a specification language for representing assertions, predicates, and conditions used to statically characterize sets of program states, invariant relationships, and requirements;

- a program design language for representing detailed procedural logic and data structures of the program; and

- a documentation format to support organization and management of the design process as well as providing the essential communication of the design to those external to design processes.

No specification language has yet been proposed for usage with the methodology. There is a great deal of research required before any such commitment should be made. Candidate languages are under investigation: SPECIAL [ROUB77], SREM [D4], AXES [HAMI76], and algebraic specifications [GUTT77]. Included in the specification language are:

- portions of first-order predicate calculus augmented by notations for abstract predicates and their characterization in more primitive terms,

- notations for primitive data types and their properties augmented by mechanisms for describing abstract types in terms of the primitives, and

- performance information including space and time complexity measures.

In the absence of specific notations processable by machine and usable by software designers, it appears satisfactory, for the present, to use a mixture of mathematical and natural language representation of these specifications. The designer, like the mathematician, may use such notations without compromising on exactness.

The WELLMADE program design language is based upon the guarded-command set introduced by Dijkstra [DIJK75 DIJK76]. Thus, the predicate transformation properties of these constructs are basic to the constructive approach of software design.

DOCUMENTATION DESCRIPTION

The overall documentation structure is presented in figure 4. The first part under the title REQUIREMENTS INTERPRETATION is the English text describing the problem to be solved. Usually the originator of the problem supplies the requirements in some form. The content of this chapter is the designers replay of these requirements. The style of the text and the language must be suitable for clear understanding by the problem originator.

The remaining part of the documentation bound by the two key words SYSTEM name and END (see figure 4) is the technical design specifications. The text following the title SYSTEM name is the table of contents of the whole document. The structure of this table of contents is:

<machine name 1>{text}

<machine name n>{text}

The text following each machine name in the table of content is an optional plain language description of the corresponding machine.

The rest of the documentation is made up by a set of documents describing the various machines. Each set is bound by the key words MACHINE<machine name> and END<machine name>.

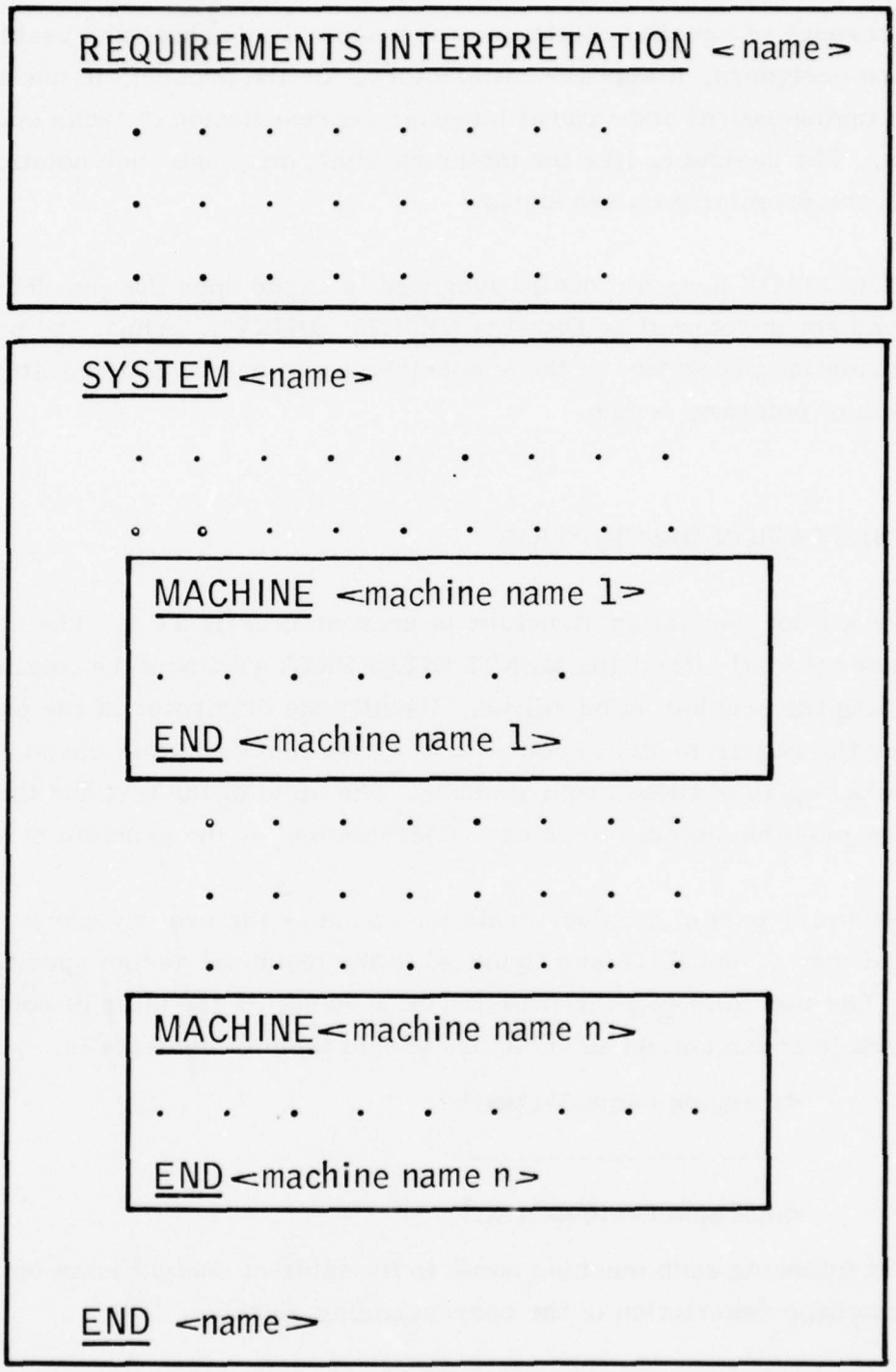


Figure 4. Overall Documentation Structure

The structure of the documentation for a generic machine is presented in figure 5. For each machine two major chapters are identified: MISSION DESCRIPTION containing the description of the machine from the point of view of the outside world, and DESIGN DOCUMENTATION containing all the necessary information to implement and to verify the correctness of the design of the machine.

The MISSION DESCRIPTION is further divided into three parts. The FUNCTIONAL SPECIFICATIONS is a description of the functionalities the machine is supposed to supply. This description is given in English and must not supply information about how the functionalities are realized in the actual design. The USAGE INFORMATION part contains all the necessary information for the actual use of the machine. At the top level it may be a draft of the system user's manual. The ACCEPTANCE CRITERIA part is the description of procedures for the acceptance of the product by the customers, if any. Typically any performance limitation or requirements will be described in this part.

In the whole MISSION DESCRIPTION chapter only FUNCTIONAL DESCRIPTION is required. Note that this chapter often is written after some design work has been done and some of the Design Documentation is written.

In the DESIGN DOCUMENTATION chapter all the technical information relative to the design of the machine are included. This chapter is written mainly by using formal notation although English text can be added for clarification. The proposed notation is presented in Appendix A and its syntax is described in Appendix B.

In the design documentation the section DEFINES contains a list of the names of the abstract data types refined in this machine. For example, with reference to figure 3, the list in machine 2 will contain one (or more) type names which are defined in machine 1. The section OVERVIEW is a plain English description (optional) of the design.

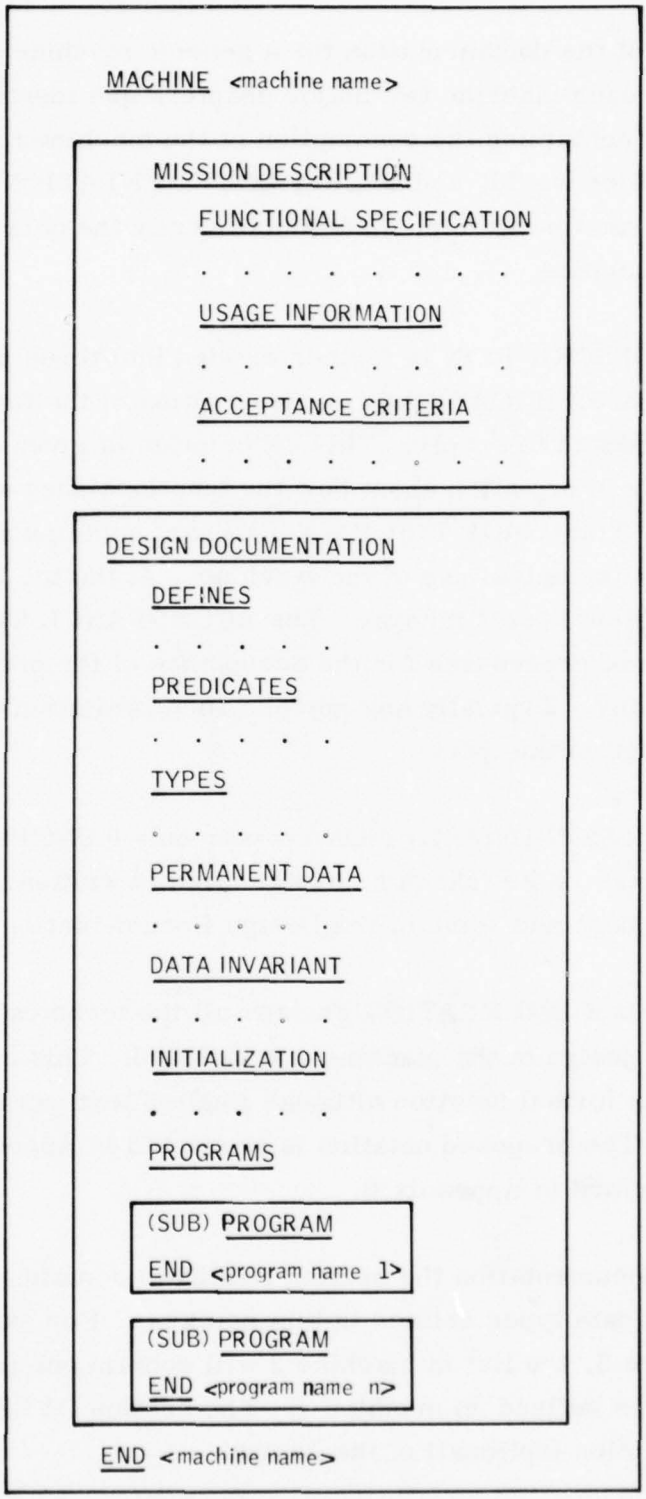


Figure 5. Machine Documentation Structure

The section PREDICATE, when it exists, contains the definition of any predicate which is convenient to define once and then use in the specifications or invariants. For example, a predicate SORTED for an array can be defined here in detail and subsequently used only by writing the verb SORTED (and the appropriate parameters). The definition is done by using English and/or predicate calculus.

The section TYPES contains the definition of all the type needed in this machine to insure the correctness of the programs. These types are supposed to be defined in lower level machines or are primitive types. The definition of types is done using the notation shown in Appendix A.

The section PERMANENT DATA defines the data that may be used in this machine by all the programs. The definition of data is done according to the notation (as shown in Appendix A):

<object id>:<type expression>{<text>}

where <type expression> is either a primitive type or a type name included in the TYPES section of this machine.

The section DATA INVARIANT contains the specifications of the only permissible input and output states for all the programs of the machine. Further restrictions can be defined as input/output specifications of specific programs.

The section INITIALIZATION contains the design of a program which is used to set up data values in such a way that the data invariant restrictions are respected.

The section PROGRAMS is the list of the names of the only programs designed for this machine. A short English text can be used to supply information about the programs.

The section(s) bound by the key words (SUB)PROGRAM and END <program name> are the documentation of the design of the programs

listed under PROGRAMS. The prefix SUB is used when the design documentation refers to a program which is not directly usable in the machine but it is called by some other program in the machine. For example, with reference to figure 3, the program P_{33} in machine 3 is a subprogram because it is used only by the programs P_{31} and P_{32}.

The documentation of programs has a structure as shown in figure 6. The whole document is enclosed between the title identified by the key word (SUB)PROGRAM and the key word END<program name>. The title specifies, if it is needed, the list of parameters that must be passed for the execution of the program. Also, if the program returns a value, the name of the return parameter will be written after the key word RETURNS.

The section OVERVIEW is an English description of the program design.

The section VARIABLES contains the declaration of all data which are not permanent data of the machine. These data must be of a type declared in the TYPES section or of a primitive type. The data declared here are considered as initialized and annihilated by the program (unless it is a subprogram) at every execution. In the case of subprograms, variables of the main program may be considered global and therefore are not initialized or annihilated at every execution of the subprogram. The declaration of variables is done according to the notation presented in Appendix A.

The section SUBPROGRAMS is the list of the subprogram names used by this program if they exist.

The sections INPUT SPECIFICATIONS and OUTPUT SPECIFICATIONS contain the restrictions on the state space which define the function of the program. The complete specifications of the input and output states is the conjunction of the specifications written in this section and the data invariant of the machine. The specifications are written in whatever assertion language the designer chooses (usually predicate calculus and English).

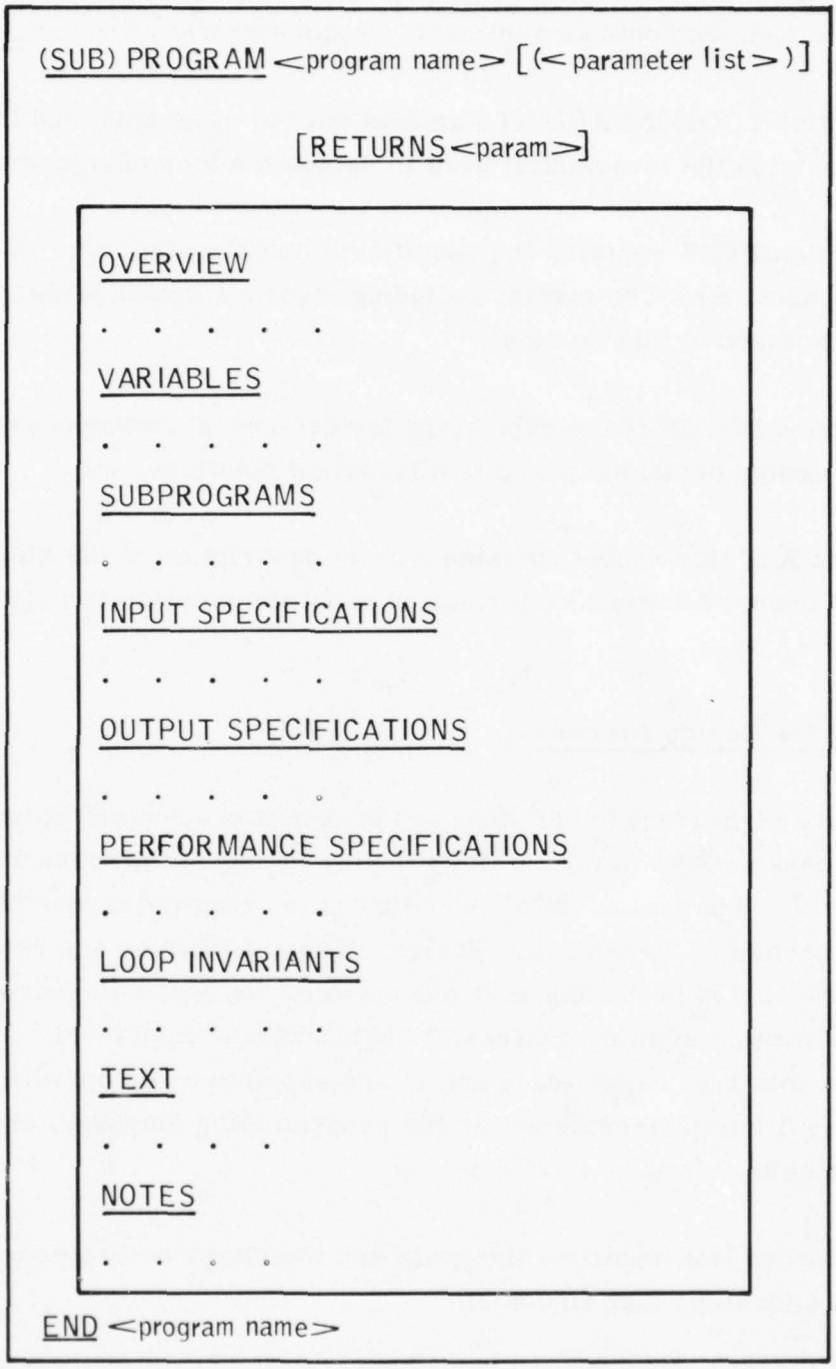


Figure 6. Structure of Program Documentation

The section PERFORMANCE SPECIFICATIONS contains any specific performance requirements as applicable to this specific program.

The section LOOP INVARIANTS applies only to programs containing loops. In this section the invariant(s) used to derive the loop design are shown.

The section TEXT contains the specification of the algorithm in p-notation (see Appendix A). Comments, including rigorous proofs when requested, can be included in this section.

The section NOTES is the repository for designers' remarks as rationale for certain design decision, possible alternative solutions, etc.

Appendix A of this report contains a brief description of the significant notation used. Additional information about the notation is in [DIJK76].

Steps of the Design Process

There are eight identifiable steps in the design procedures of the RDM. These steps and the flow of activity during the design process are displayed in figure 7. The major set of activities is for identifying and designing a single machine of the software design. These activities are repeated in a top-down fashion producing abstract machine designs with decreasing degrees with decreasing degrees of abstraction at each level. When all machine data type capabilities are either supplied by an operating system or are easily implemented within the programming language, the design process halts.

The following list identifies the goals and the major activities of each of these design steps (see figure 7):

Requirements interpretation -- The goal of this step is to identify and interpret a set of requirements which have been supplied by the customer

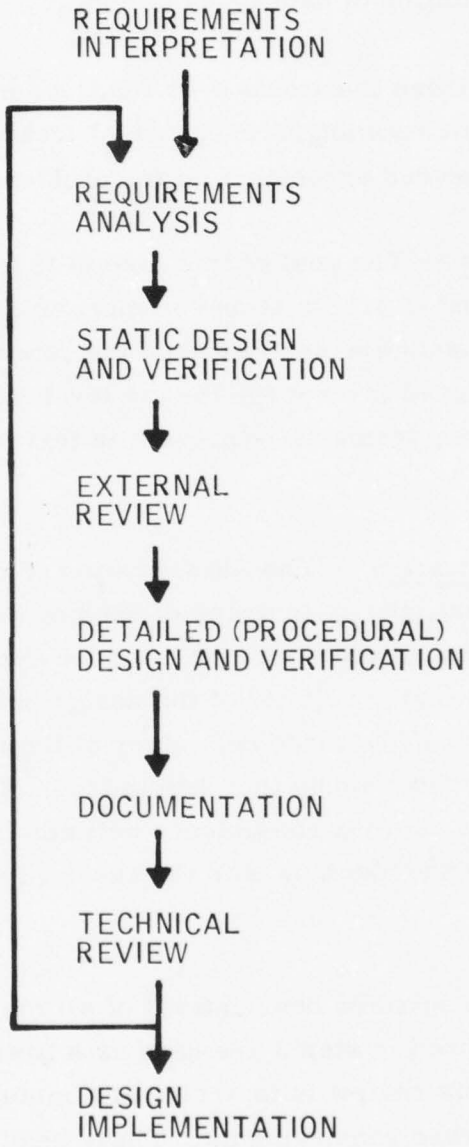


Figure 7. Design Procedures

requesting the software design. These requirements must be translated to the functional specifications of the software system. During this process some preliminary definition of data types is done.

The attempt to define these functional specifications may produce information necessary for some meaningful exchange of ideas with the customer in order to produce the needed agreement on the problem requirements.

Requirements analysis -- The goal of this step is to analyze the set of functional specifications and to arrive at one or more machine identifications. The machine requirements are derived from the data type specifications of machines already designed (except for the top level machines). The interpretation of machine requirements is present in text form as an external mission description.

Static design and verification -- The identification of a machine's data state space, its data invariant, the state space of each of its programs, their input/output specifications, and their performance specifications are done in this step. This is a major activity of the design process and results in the static specifications of the machine. Many of the machines capability requirements are identified during this design step. The hierarchy of machines is verified by showing consistency between invariant and input/output specifications of the machine with the capability requirements of the upper level machine.

External review -- The mission descriptions of all machines as prepared in step 2 and possibly refined in step 3 are used as a basis for customer review. The goal of this review is to arrive at a mutual agreement of all interpretations of software requirements. Thus, results of the external review are either to revise requirements and to backtrack to the appropriate step, or to freeze the capability specifications from which the mission descriptions have been derived. Note that external reviews are not necessary for every machine design. They usually occur only at preliminary design stages (the higher levels in the hierarchy).

Detailed (Procedural) design and verification -- The constructive approach combining Dijkstra's discipline for program design with stepwise refinement (when it is needed) is used to construct a detailed program design. Note that the design discipline is guided by the verification technique. Thus verification becomes part of the detailed design process. Syntax checks and type consistency checks are part of this design step and automated tools should be used at this end. If formal verification is required, it must be supported by automated tools like weakest-precondition generators and theorem provers. An integral part of this design step is the identification of operations on data types. Note that this step may affect the static machine design. In fact, the whole design process requires some iteration between steps 2 and 5. This point will be clarified in the next sections. Finally, static performance assessment, when required, is a major activity of this design step.

Documentation -- Although documentation is part of the previous two steps of the design process, the goal of this step is to integrate all parts of the design documentation into a single document and to update the mission description, if necessary. The mission description must conform to both the design and the capabilities from which it is derived. Documentation involves the addition of designer notes, comments, and design overviews.

Technical review -- The design document and the mission description become the primary vehicle for technical review. The results of this review are to either freeze the machine design or to cause the process to backtrack to earlier steps in order to repeat the design so as to correct any error.

Design implementation -- The result of the design process is a document which represents a software product design. The design process for any part of the software is completed when the module's required capabilities are easily implemented and all module documents have been frozen by the review process.

Note that these design procedures do not preclude the possibility of parallel design activity. The machine design activities may proceed in parallel under the proper supervision of the design process.

WORK PROCEDURES

The identification of the above eight steps of the design process of an abstract machine supplies a framework for the organization of the work. To complete the description of the methodology, however, it is necessary to identify generalized work procedures in such a way that a form of mental discipline will result.

Unfortunately because of the highly creative aspects of the design task, work procedures can be only guidelines rather than precise prescriptions. These guidelines are based both on experience and on the principles of constructive design.

The procedural steps identified in the previous section are now defined in terms of work output expected from each activity. This output is usually some portion of the documentation completed. This permits a factual managerial control of the progress of the design activity.

Requirements Interpretation

The purpose of this step is to understand the particular design problem. The requirements are usually given by the customer in some imprecise form and an agreement on their interpretation must be reached between the customer and the designer. Since the customer is not necessarily an expert with notation used in methodology this agreement must be reached on the basis of common language. When the agreement about the interpretation of requirements is reached, the document REQUIREMENTS INTERPRETATION is prepared and the text discussed in a kick-off review.

Requirements Analysis

On the basis of the requirements interpretation which has been agreed upon, the overall functionalities of one or more abstract machines must be established. During this process any requirement of permanent data storate (if any has been identified) as well as the identification of functions this machine is required to accomplish, will be considered, and a first cut of the machine state space definition and of the specifications of the program completed. For machines of intermediate levels, the requirements are the type requirements of other machines.

A tentative mission description of the machine is the formal documentation output of this step.

For machines at intermediate levels, at least an informal verification of the correctness of the representation of the upper level data types by the machine should be performed. A more formal verification can be made according to the methods described in [HOAR72a] and in [WULF76]. The output of this formal verification, when it exists, will be recorded in the section Notes.

Static Design

The informal identification of storage and functionality is now formalized. In this phase the machine permanent data, data invariant, the specifications of the program and the necessary data representations of the upper level data are formalized by using the RDM notation (see Appendices A and B). Also, some abstract data types may be identified. This will produce the description of the data type operations requirements which are those for other lower level machines. The mission description will be updated and all the determined information will be recorded in the appropriate sections of design documentation.

In the next section, some practical suggestions for devising and checking the static specifications are described. These suggestions should be kept in mind and used before the external review.

External Review

At this stage it is useful to check with the customer that the design is proceeding in the right direction. The means for performing this check is the documentation and, more specifically, the mission description for non-technically oriented persons and the results of statis design for technical people.

An approval from the customer must be obtained at this point or a backtrack to an appropriate position is required. The documentation of findings of the review is the output of this step.

Procedural Design

The machine state space and the specifications of the various programs are now known. It is thus possible to design and prove all the programs by using the constructive approach. (The constructive approach is described in detail in the next section of this report.)

The design of the various programs can be done simultaneously by different designers. However, a coordination of the designs by a single responsible person is indispensable. During the design of the program, additional abstract data types and/or operations on abstract data types are usually needed. The documentation of the machine will be updated with the newly defined operations. At the end of this step the documentation of the machine and of all the programs for the machine is complete.

Internal Review

After completion of the procedural design, the design process must be verified by competent persons. The most formal of this verification is the use of an a-posteriori proof process to verify the program. Note that all the necessary ingredients for the proof process are available if the designer had used the appropriate level of formalism. Formal proofs will be recorded under the Notes section for each program of all machines.

Other techniques may be used in most cases with effectiveness. One of the most effective is possibly the presentation of the design by the designers

to a small group of colleagues. The review may point out faults in the design and in that case the errors must be corrected and another review passed.

PRACTICAL SUGGESTIONS FOR STATIC DESIGN

It is useful to view specifications as three types of relations among the state space variables [GERH76]. The first type includes all the possible assertions that can be imagined about input states. These can be stated independently from output. The second type includes the assertions about output that can be stated independently from input. The third type includes all the assertions that can be considered as relations between input and output.

The following suggestions are helpful in devising specifications from requirements:

- Check that all assumptions, often tacit in the expression of requirements, are made explicit.

- Structure the specifications by using abstract predicates in such a way that the formal specifications read similar to the informal statements of requirements. This process may require several steps from a very vaguely expressed concept to a rigorous formal specifications expression.

- Test the specifications. This does not mean to put some numerical value in the variables and check the truth values of the predicates. Useful tests are a) try to find an absurd program that satisfies the specifications. Success of this test indicates incompleteness. b) Break the specifications in cases and test them against the informal requirements. c) Formulate specifications in a different form to show consistency of the previous two forms. Find an independent

verifier with appropriate competence, and have the formal specifications translated back in informal statements.

THE CONSTRUCTIVE APPROACH

The constructive approach is the core of RDM. The idea is due to Dijkstra [DIJK76] and is used to derive correct programs. Although not shown in the examples of this section, it is essential, whenever necessary, to define new operations on abstract data types or even new data types in order to contain the size of the program and the difficulty of the proof within manageable proportions. This produces new levels of abstract machines whose functions will be realized by programs constructed by using the same process.

The theory has been already outlined in the "Constructive Design Principle" section of chapter 2. The semantics of the algorithm constructs together with the loop invariant theorem allow the construction of correct programs from the static specifications. The following three simple examples are presented to illustrate the approach. The first shows how to use the selection construct while the second and third illustrate the loop invariant theorem.

First Example: Selection of the largest of two numbers -- Let us suppose we are required to write a program which will give to a variable Z the greatest value of two numbers X and Y [DIJK76]. The output states specification is: $R = (Z \geq X)$ and $(Z \geq Y)$ and $(Z = X$ or $Z = Y)$
By using the assignment $Z := X$ the postcondition $Z \geq Y$ is obtained if the weakest precondition is (see 2.4)

$$wp(Z := X, Z \geq Y) = X \geq Y$$

Analogously for the assignment $X := Y$:

$$wp(Z := Y, Z \ X) = Y \geq X$$

By remembering the definition of the wp(if...fi, R) in chapter 2, "Constructive Design Principle" and making B1=X≥Y and B2=Y≥X the following program can be written:

$$\underline{\text{if}}\ X \geq Y \rightarrow Z := X \ \blacksquare\ X \leq Y \rightarrow Z := Y\ \underline{\text{fi}}$$

which furnishes correct results when:

$$\text{wp}(\underline{\text{if}}\ X \geq Y \rightarrow Z := X \ \blacksquare\ X \geq Y \rightarrow Z := Y\ \underline{\text{fi}}, R) =$$
$$(X \geq Y\ \underline{\text{or}}\ X \leq Y)\ \underline{\text{and}}\ (X \geq Y\ =>\ (\text{wp}(Z := X, Z \geq Y) = X \geq Y))$$
$$\underline{\text{and}}\ (Y \geq X\ =>\ (\text{wp}(Z := Y, Z \geq X) = Y \geq X)) = \underline{\text{True}}$$

Therefore the program is correct for any input state in the state space of a machine where X, Y, Z are integers. Note that the non-deterministic construct in the case X=Y will assign (correctly) either X or Y to Z.

Second Example: An integer division program -- In this example the use of the loop invariant theorem is shown. Suppose we are required to write a program calculating the quotient Q of two integers X (the dividend) and Y (the divisor). Let us also assume that the input states specification as:

$$X > 0\ \underline{\text{and}}\ Y > 0.$$

The determination of a satisfactory output states specifications is not a totally trivial task. However, by reasoning about the problem as follows, a precise definition can be reached.

The first assertion that comes to mind (to specify a division between integers) is:

$$X = (Q * Y) + R$$

(where R is the remainder of the division) which is simply a restatement of the division process in terms of multiplication. While the above statement is obviously a requirement for the outcome of the program, it should by no means represent the reader's intuitive feel for division. For example, if Q is assigned zero, R=X would satisfy the above predicate but, except for the case where Y>X, does not represent the notion of division. A further restriction could be:

$$R < Y$$

which states that the remainder after division is expected to be less than the divisor (if the divisor is greater than 0, which is guaranteed by the input assertion $Y>0$). But not even the conjunction of the two predicates above is still a sufficiently precise definition of the outcome of division, because by putting $X=10$, $Y=3$, $Q=4$, and $R=-2$, for example, the specifications become:

$$10 = 4*3+(-2)$$

satistying both statements and yet still violating the known rules of division. The remainder after the division is expected to be greater than or equal to zero. By adding this final restriction the following output states definition which defines the desired effect of the division program is obtained:

$$(X=Q*Y+R) \text{ and } (0 \leq R < Y)$$

Inspection of the input and output states specifications shows that while the postcondition is a rather strong constraint on the possible output values, the precondition is considerably weaker. In fact, only a single set of values at a time will satisfy the postcondition, while any pair of positive non-null integers (an infinity of values) will satisfy the precondition.

The final program will be one, or more probably, several constructs concatenated into a sequence which will produce an ordered series of states definitions, the first one equal to (or containing) the input states definition, the last one equal to (or contained in) the output states definition. In order to build the sequence of constructs (that is, to solve the design problem), it is possible to start from the input states and postulate some intermediate state which can be obtained from the input, then use this new intermediate state as input and build another state...and so on, until a state equal to (or contained in) the output state is obtained. An alternative method is to work backwards, starting from the output, defining an input which in turn becomes a new output ... until a state equal to (or containing) the given input state is defined. A first consideration is indicated by the use of the words "contained" and "containing" for forward process and for the backward process, respectively. In more precise terms this means that in the forward process it is needed to determine, at every mental step, a construct which will eventually bring us to the determination of a condition equal to or

stronger than the given postcondition, while in the backward process it is required to reach a condition equal to or weaker than the given pre-condition.

A strong condition, by its nature, contains more information than a weak one. Consequently it seems easier to imagine a weaker condition out of a stronger one (it can be viewed as a removal of information), than to create a stronger condition out of a weaker one (which requires the creation of new information). Now attempt the apparently easier backward approach. By examining the output state definition, it is possible to assume that an intermediate state defined by:

$$(X=Q*Y+R) \text{ and } (R \geq 0)$$

has already been reached, in some way, from a state satisfying the pre-condition. The reasoning leading to the choice of the above intermediate state is highly subjective. In fact, it is made because of some imprecise knowledge about the algorithm. It is known that if Y is subtracted from X, and a count of how many times it can be done without creating a negative remainder is maintained, the count will become equal to the quotient. It seems reasonable then to remove from the postcondition the limitation $R<Y$, and try to solve the problem of moving from a state defined as above to the state which does respect the postcondition. But it must be pointed out that this is not an automatic choice: it represents a highly creative part of the design process. Another state definition could have been chosen as the intermediate state, for example, the one defined by:

$$(X = Q*Y+R) \text{ and } (R<Y)$$

and then attempted to make $R \geq 0$ in order to obtain the solution. However, any attempt to build a program according to this choice would probably demonstrate the choice as a poor one. Going back to condition $(X = Q*Y+R)$ and $R \geq 0$) as the choice for an intermediate state, it is immediately seen that the condition allows $R \leq Y$ (as well as $R>Y$). If $R \leq Y$, it seems reasonable to attempt to reach the output state by subtracting Y from R. In so doing, if Y and R are positive, a step toward the post-condition will be made but it will violate the truth of the first term of the

intermediate condition, namely $X \neq Q * Y+R$. In order to re-establish the truth of $X=Q * Y+R$ after R has been decreased by Y, it is sufficient to increase Q (again supposed a positive number) by one. We have therefore determined that the following program segment:

$$R := R-Y;\ Q := Q+1$$

when initiated in a state satisfying $X=Q * Y+R$ and $R \geq 0$ will terminate in a state satisfying the same predicate and will make a step toward the desired output state. But the desired output state is reached when $R < Y$, therefore it is intuitively clear that the following loop is the appropriate solution:

$$\underline{\text{do}}\ R \geq Y \rightarrow R := R-Y;\ Q := Q+1\ \underline{\text{od}}$$

provided that R, Q, and Y are positive numbers.

It is now very simple to determine the next segment which will accept any input state that satisfies the required precondition.

Remember that all of the above reasoning thus far required that R, Q, and Y are positive numbers. This is perfectly acceptable, since the post-condition does not specify anything about the sign of either Q or R, while the precondition does require Y to be positive. Thus at loop initiation there is the requirement for states satisfying:

$$(X=Q * Y+R)\ \underline{\text{and}}\ (R \geq 0)\ \underline{\text{and}}\ (Q \leq 0)\ \underline{\text{and}}\ (Y \geq 0)$$

By choosing $Q=0$, the term $X=Q * Y+R$ imposes $R=X$. Thus the following program segment will create a state satisfying the above condition when activated in a state satisfying $X \geq 0$ and $Y \geq 0$.

$$Q := 0;\ R := X;$$

Since $X \geq 0$ and $Y \geq 0$ is implied by $X > 0$ and $Y > 0$, the design task could be considered successfully completed. However, termination has not yet been shown. At this end the function C (see "Constructive Design Principle," chapter 2) must be found and it must be shown that it is properly handled by the program under the defined conditions. A good choice for C appears to be $C=R$. In fact, R is a function of the state variable which has the value $X > 0$ at the input and at every cycle is reduced by the positive

non-null value Y, i.e., in which case the loop terminates if $Y>0$. Thus, if a precondition $X\geq 0$ and $Y\geq 0$ as determined by the preceding reasoning is accepted, we do not satisfy the termination criteria (C would be null at loop initiation for $X=0$ and would not be reduced at every cycle for $Y=0$). Therefore, the conclusion that the weakest precondition $X\geq 0$ and $Y>0$ is required can be reached, which implies the required input states specifications, and that the following program is totally correct:

```
Q:=0; R; X;
do  R≥Y  →R:=R-Y; Q:=Q+1 od
```

In light of the invariance theorem introduced in the preceding chapter, we recognize immediately that the intermediate states definition is the loop invariant. The process thus can be reinterpreted as follows. Suppose that, based on some informal definition of the algorithm the need for an iteration is determined. Then the postcondition must be broken down into two parts: The invariant, P, and the negation of the loop guard, non B. The core of the loop is then designed as the one that maintains the invariance of P and that strictly reduces the termination function C (by a reasoning similar to the one used above).

From the above, a generalized scheme for the design of loops can be outlined. First, break down the required postcondition and define an invariant. Design the loop core as a program segment maintaining the invariant and strictly reducing the appropriate termination function. Complete the loop with the guard and check the values of the termination function at loop initiation and at termination (when the guard becomes false). Determine an appropriate program segment establishing the invariant from a state "closer" to the required input (possibly equal to the precondition). The above process is not a fixed rule. It appears to be a good practice, but it is not necessarily the only way to design loops. However, if the programmer is able to put his or her design in the above form, regardless of how he or she constructed the program, a proof of total correctness is obtained. In actual practice, the process will proceed largely by repeated trial and

error, based at each try on an attempt to define an invariant relation and a termination function in terms of some suitable abstract data types.

It must be re-emphasized that this approach avoids consideration of program execution. The program is always regarded as a mechanism which may exist in different states and those states are defined by relations among the variables, never by the actual values. This way of looking at programs permits effective application of the above design process. A final observation concerning the example above relates to the use of the invariance theorem.

The invariance theorem permits definition of a dynamically changing phenomenon, as the loop, in terms of a static relationship. The loop invariant relation is the only criterion needed to show the correctness of the loop. This allows the construction of loops without reference to dynamic program behavior. Together with the termination function, which also does not require consideration of program execution, the invariant is the tool which permits the handling of iteration constructs in such a way that the overall effect of a loop is solely defined by the input and output states.

Third Example: a non-deterministic program -- With this example a program whose design calls for the non-deterministic constructs, even though the requirements are strictly deterministic, will be presented. Suppose a program design is required that, given a non-empty array of objects, A, of a certain type, say integers, will rearrange the elements of the array in such a way that the array will be partitioned in two distinct portions; the lowest one containing values less or equal a given value, V, and the highest one containing values greater or equal V. Because of the way the requirements have been expressed, they are actually not deterministic; any of the elements equal to V, if they exist, can be placed in either one of the two portions of the partitioned array. To respect the claim that the program's requirements are strictly deterministic, the additional restriction that all the elements of A are distinct and not equal to V must be imposed. The program for the latter case will be first developed and the solution for the other case will be shown.

According to the procedure of RDM, first the requirements must be made more precise. From the requirements it is not difficult to obtain the following output states definition:

$$\{\forall i;A.lob<i<H:A(i)<V \text{ or } H=A.lob\}$$

and

$$\{\forall i:k<i\leq A.hib:A(i)>V \text{ or } K=A.hib\}$$

and

$$\{H>K\}$$

and

$$\forall i:A.lob\leq i\leq A.hib:\text{not } \exists j:A.lob\leq j\leq A.hib:$$

$$[(i\neq j \text{ and } A(i)=A(j)) \text{ or } A(j)=V]$$

The first term of the conjunction defines the lower portion of the partitioned array as the one between A.lob and H-1 included. The case of an empty portion is represented by H=A.lob. The second term defines the higher portion of the partitioned array as the one between K+1 and A.hib. The case of an empty portion is described by K=A.hib. The third term tells us that the two portions are adjacent (notice that if one of the two portions is empty this term represents the fact that the other portion's index becomes less than A.lob or greater than A.hib). The fourth one expresses the fact (which is supposed to be verified in input) that all the elements of A are distinct and not equal to V. This last term is needed to avoid to make the third term a contradiction of the first and the second. (If there is one element equal to V it must be placed somewhere in the array, therefore the final arrangement in the array must contain a zone where A(i)=V are placed. Consequently, K cannot be less than H.) In input, the fact that all the elements are distinct and not equal to V must be required. That is:

$$A.dom\geq 1 \text{ and}$$

$$\forall i:\ A.lob\leq i\leq A.hib \Rightarrow A(i)\neq V$$

and

$$\forall i,j:\ A.lob\leq i,\ j\leq A.hib \Rightarrow (i\neq j \Rightarrow A(i)\neq A(j))$$

It is clear by inspection of the states definitions there is a need for an iterative procedure. This requires the definition of an invariant. To construct the invariant from the output states definition, imagine a state whereby the partitioning of A has been partially accomplished. That is, there is a portion of A, between the indices H and K where values greater or smaller than V are still mixed. This fact is expressed by:

$$\{\forall i: A.lob \leq i < H: A(i) < V \text{ or } H = A.lob\}$$

and

$$\{\forall i: K < i \leq A.hib: A(i) > V \text{ or } K = A.hib\}$$

and

$$\forall i: A.lob \leq i \leq A.hib \Rightarrow A(i) \neq V$$

and

$$\forall i, j: A.lob \leq i, j \leq A.hib \Rightarrow (i \neq j \Rightarrow A(i) \neq A(j))$$

The invariant is identical to the output states definition with the exception of the term $H > K$. Assuming this term denied, namely $K \leq H$, if the element $A(H)$ is less than V, then the statement:

H:=H+1

preserves the invariant. Analogously, if $A(K) > V$, K:=K-1 preserves the invariant. In the case $A(H) > V$ the sequence of statements:

A:swap(H, K) ; K:=K-1

preserves the invariant. In fact, the swap operation will make $A(K) > V$ and $A(H) > V$ or $A(H) < V$. Thus the portion bound by K can be increased by one (K:=K-1). Similarly, if $A(K) < V$:

A:swap(H, K) ; H:=H+1

preserves the invariant. At this point the following program can be written:

do K≥H cand A(H)<V→H:=H+1

K≥H cand A(H)>V→K:=K-1

```
        K≥H cand A(H)>V→A:swap(H,K);K:=K-1
        K≥H cand A(K)<V→A:swap(H,K);H:=H+1
    od
```

which will transform any state satisfying the invariant in one satisfying the required output state, if it terminates.

Termination proof is rather simple. Assume for termination function C=K-H+1. This function is positive (C≥1) if K≥H, and it is strictly reduced by one by any of the guarded commands. It cannot ever become negative if the loop is initiated with K≥H.

To complete the program H and K must be initialized in such a way that the invariant is established. The final program is then:

```
H,K:=A.lob,A.hib;
do K≥H cand A(H)K<V→H:H+1
   K≥H cand A(K)>V→K:=K-1
   K≥H cand A(H)>V→A:swap(H,K);K:=K-1
   K≥H cand A (K)<V→A:swap (H,K);H:=H+1
od
```

If the assumption that all the elements are A are distinct and not equal to V is removed, the following program is obtained:

```
H,K:=A.lob,A.hib;
do K≥H cand A(H)≤V→H:=H+1
   K≥H cand A(K)≥V→K:=K-1
   K≥H cand A(H)≥V→A:swap(H,K);K:=K-1
   K≥H cand A(K)≤V→A:swap(H,K);H:=H+1
od
```

Notice that A.dom=0 is permitted.

PERFORMANCE REQUIREMENTS

The starting point of program design activity is the gathering of the appropriate set of functional and performance requirements. The functional requirements are statements describing what the program is supposed to do. They are transposed into more exact specifications and ultimately into algorithms and data structure descriptions by using the techniques described in the previous reports.

It is possible to imagine the performance requirements as a set of limiting conditions under which the required functionality is to be realized by the design. These limiting conditions may be given in the form of cost limitations or more appropriately in the form of computational resources limitations (clearly, if given in the latter form, the transformation in terms of cost is a straight-forward one).

Every computational resource can be characterized by its capacity in terms of storage and by its speed of performing the given task. We can classify the performance requirements as storage space and execution time requirements.

Space requirements are usually given as a simple number representing the upper limit of the specific storage resource. In the case of execution time limitations, the use of an upper limit may not be very significant.

This is due to the extreme variability of program time performances with the input data configuration. This fact makes it impossible to express

performance requirements by some absolute number; it rather imposes the expression of the requirements as functions of input data configurations. However, the descriptive power of a single number is too attractive to be discarded. Consequently, so called best case, worst case and average performances are often considered and requirements could be expressed in those terms.

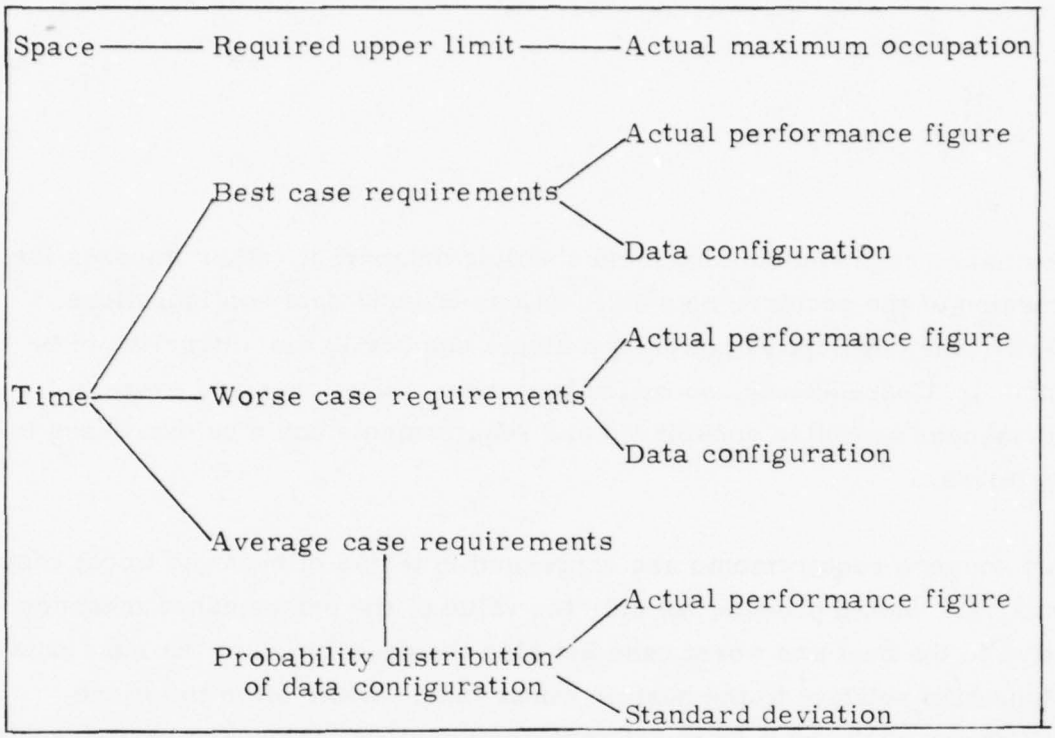
If performance requirements are expressed in terms of best and worst case the analysis should produce not only the value of the performance measure relative to the best and worst case but also the description of the input data configuration relative to the best or worst case. While often times the identification of the best or worst cases are obvious, there are cases requiring considerable work for their determination.

The average case is even more difficult to define. Some probability distribution of the input data configurations must be assumed. This is not a simple assumption to make and considerable uncertainty exists about the value of an average calculation, perhaps laboriously performed, on such uncertain assumptions.

Table 2 presents a recapitulation of the various forms of performance requirements as defined above.

Another general consideration about static performance is that the evaluation of performances is an "after the fact" operation. That is, given a program design it is possible to evaluate the actual time performance figure and the storage occupation.

Table 2. Performance Requirements



This is in contrast with the way correctness considerations are used in the Rational Design Methodology, where they are assumed as basis for the development of the program. It is possible to state that, in general, the program is developed first from its functional specifications under the criterion of correctness without consideration of limitations imposed by the performance requirements. Then the performance requirements are verified and if it is necessary, the program will be modified. The design documentation will represent the final product of the design process. Therefore, the program design which satisfies the performance requirements together with the description of the analysis needed justify the design. It is often useful to maintain the design of the program with no limitations due to performance for possible different implementation or because the demonstration of correctness of the final product is easier by showing the changes with respect to the original design.

Space Requirements Analysis

Storage space requirements are usually expressed in terms of appropriate units of storage in the object machine. For this reason it may happen that during the top-down development, space requirements will be expressed as limitations on abstract data types. For example, suppose that a file F has been defined:

XX:abstract

F:XXarray

and that the limitation for F to be smaller than 100 machine words is required. Then, the requirements would be expressed as:

$$SIZE(F) \leq 100 \text{ machine words}$$

$$SIZE(XX) \leq (100/SIZE(F))$$

$$SIZE(F) = F.dom * SIZE(XX)$$

where the function SIZE(W) is defined as the one that supplies the number of words necessary to store W. If a lower level development will produce the following data declaration:

yy:abstract

XX:(A:integer; B:integer; C:yy)

the space requirement will be given by the relation:

$$SIZE(A) + SIZE(B) + SIZE(C) \leq (100/SIZE(F))$$

Assuming now that the decision to represent integers by half machine word has been done, the requirement for all occurrences of type yy becomes:

$$SIZE(C) \leq (100/SIZE(F) - 1)$$

The above statement can be verified when the representation of the datum C is decided. Note the manner in which requirements verification are respected depends on the value of SIZE(F) which now can be calculated in terms of machine words. For example, if the representation of C makes SIZE(C) = 3 expressed in terms of F.dom instead of the function SIZE:

$$(F.dom/4 \leq 100) = (F.dom \leq 25)$$

The above example indicates that the analysis of space requirements is not completed (in the general case) until the actual machine representation of data is decided. However, the use of appropriate functions, as illustrated by the use of SIZE in the above example, should permit the top-down development of the program and the subsequent verification of space requirements.

It may happen that the process of verification shows the requirements are not satisfied by the design. In the previous example, suppose that the limitation on a datum of the type file, F, (100 machine words) is given as core memory limitation while the actual size of F, as given by the problem, is given in terms of F. dom and it is much higher. If the designer had assumed the whole F was contained in core memory, now the design must be changed.

It appears, however, convenient to proceed to the top-down design being concerned only (or mainly) with correctness by assuming unlimited space resources. The only concern with space requirements should be limited to definition of relations of the type shown in the above example.

Execution Time Performance

The study of the literature on the subject of program execution time evaluation, indicates three major categories covering the areas of interest:

- Computational complexity
- Analysis of algorithms
- Formal verification of program performances.

The research in computational complexity addresses the problem of finding the "best possible" algorithm that solves a particular problem. Usually there are several possible reasonable algorithms for a problem. Each one has a certain significant cost function, such as the number of computational steps as a function of the problem size, that can be associated to it.

The theory of computational complexity attempts to answer questions of this kind: What are good algorithms for the problem? Can one establish a low bound for one of the cost functions? Is the problem perhaps intractable in the sense that no algorithm can solve it in a practically feasible time?

There is no doubt about the importance of the theory of computational complexity, particularly in light of several recent achievements. From the point of view of evaluation of program performances, however, the theory appears of little use. That does not mean the designer should not be aware of the results of the theory; these results can be vital for the choice of an efficient algorithm. What is intended is that the problem of evaluating execution time on a specific program design is not within the scope of the theory. An excellent recent overview of the subject of computational complexity is [RABI77], which also contains an extensive list of references.

Category b) represents, probably, the most useful approach for the purpose of static performance evaluation. The principal proponent of this approach is D. E. Knuth, who has presented many examples in his books [KNUTH].

The basic concept used in the time analysis of algorithm is the so called frequency analysis. This consists of the determination of how many times each part of the algorithm is executed as a function of significant properties of the configuration of input data. It is usually not difficult to determine the best or worst case, but the determination of the average (assumed a probability distribution of the inputs) is much more complicated.

In many practical cases, however, instead of assuming a probability distribution of the inputs, the calculation of frequencies can be done for several choices of significant input data configuration. In this way, a plot of frequencies as functions of one (or more) parameters characterizing the input data configuration, can be built giving a good understanding of the time performance behavior of the program.

We suggest the use of the frequency analysis technique for time performance evaluation in the RDM.

Category c) is a recent technique recently suggested by B. Wegbreit [WEGB77]. The basic idea is to write assertions about probability distribution of data configurations and, by using techniques of theorem proving, to prove that other probability assertions relative to intermediate stages of the computation hold. Because of the very limited experience of use of this approach at this time, it does not appear suitable for inclusion in a design methodology. However, the ideas presented are attractive and should be the basis for further research.

Examples of Frequency Analysis

As stated above, frequency analysis appears to be the most useful technique for time performance evaluation. We will present now the technique by showing two examples.

First Example -- The following program finds the maximum element in an array of integers, A, and stores it in M. It will also store in J the index of A corresponding to A(i)=M.

```
M, J := A.low, A.lob;  _ _ _ _ _ _ _ _        S1
I := J;  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _      S2
do I ≤ A.hib
   if A(I) ≥ M → M := A(I); J := I
   ▌ A(I) ≤ M → skip                          S3
   fi;
   I : I+1
od
```

Statements S1 and S2 are executed 1 time for every program execution. Statement S3 (the loop) requires two different numbers for the characterization of its frequency. One is relative to the guard, another to the body. If the loop body is executed n times the guard will be executed n+1 times. Inside the loop the if...fi statement as a whole will be obviously executed n times and that applies to both guards $A(I) \geq M$ and $A(I) \leq M$ (according to the semantics of the if...fi both guards must be evaluated even though the evaluation can occur in parallel).

For the two guarded command lists we define a number $a \leq 1$ which represents the percentage of n for which the first list (M:=A(I)); J:=I) is selected. Then, the number of times the first list will be executed is a*n and the second list will be executed (1-a)*n times. The results are presented in the following table:

Table 3. First Example Frequency Analysis

Program Text	Frequency
M, J := A.low, A.lob;	1
I := J;	1
do I ≤ A.hib→	n+1
if A(I) ≥ M→ M := A(I); J := I	n, (a*n)
▮ A(I) ≤ M skip	n, ((1-a)*n)
fi;	
I := I+1	n
od	

The number of n is clearly equal to A.dom. The determination of the best and worst case requires the knowledge of the cost of the two guarded command lists in the if...fi. It appears more than reasonable to assume that the cost of skip is less than the cost of the two assignments. Then, the best case is for a=0 and the worst for a=1. The best case corresponds to the case of the array sorted in descending order (J=A.lob) and the worst case corresponds to the array sorted in ascending order, (J=A.hib).

In order to give a limit about the difficulty of intermediate cases determination, let us consider the case when the maximum is at an I=A.lob+(A.dom/2). There is no doubt that for half of the array after the maximum has been found only the second guarded command list will be executed. However, that does not allow the conclusion that a=.5. In fact, a can be any value depending on the arrangement of elements in the first half of the array. It would be possible a best sub-case where the elements in A are arranged in decreasing order up to and including the one in J-1 and a worst sub-case where the elements were in reversed order. The only way to reach any conclusions is to assume either specific data configurations of interest or a probability distribution. The latter case is extensively treated by D. E. Knuth in [KNUTH] Vol. 1.

Second Example -- The following program partitions a given array A in two portions: one containing all the elements not greater than a given value V and the other containing all the elements not less than V.

```
H, K := A.lob, A.hib;
do H ≤ K →
   do A(H) < V → H := H 1 od;
   do A(K) ≥ V → K := K-1 od;
   if H ≤ K → A: swap(H, K); H := H+1; K := K-1
   ▌ H < K → skip
   fi
od
```

The frequency analysis of this example is more difficult than the previous one. In fact, the example is presented to show the handling of nested loops:

Table 4. Second Example Frequency Analysis

Program Text	Frequency
H, K := A.lob, A.hib;	1
do H ≤ K →	n+1
do A(H) < V →	x+n
H := H+1	x
od;	
do A(K) > V →	y+n
K := K-1	y
od;	
if H ≤ K → A:swap(H, K);	n
H := H+1;	n
K := K-1	n
▮ H > k → skip	n
fi	
od	

The analysis shows that the number of comparisons of A(H) or A(K) with V is equal to A.dom. That is:

$$X + Y + 2n = A.dom$$

The only additional interesting figure to determine is n, the number of "swap" operations. The result is known from the literature [WIRTH] and it is

$$n \tilde{=} A.dom/6$$

The reasoning for the determination of n is as follows. Let x be the value of a generic element in A, and let us assume that m ($m \leq A.dom$) elements in A

are less than x. Then the probability that the element with the value x will be exchanged is

$$A.dom - m + 1 / A.dom$$

The expected value of n is obtained by summation of all the possible choices of m and dividing by A.dom:

$$n = \frac{1}{A.dom} \sum_{m=1}^{A.dom} \frac{A.dom - m}{A.dom} (A.dom - m + 1) =$$

$$= \frac{A.dom}{6} - \frac{1}{6\ A.dom} \cong A.dom / 6$$

This example indicates, also, how to proceed for frequency analysis of a program developed in various levels of abstraction. Assuming that the swap operation is not a primitive one, the determination of the number n is to be used to determine the frequency of the more elementary operations' in the program realizing swap. For example, if swap is realized by the following text:

```
T := A(H); A(H) := A(K); A(K) := T
```

The frequency analysis shall show that all the three assignments are executed n times.

SUMMARY OF THE DESIGNERS ACTIVITY

The suggested design procedure for software systems as reflected in the designer's activity is described briefly below.

1. Translate the requirements into a static specification. In doing so the data types must be partially defined (as a name and a set of values).

2. Design one or more programs to reflect the static design from 1. In so doing, the necessary operations on types are discovered and the type definitions may be completed.

3. Formally document the work done thus far, check for consistency, and correct as necessary. Formal proofs of algorithms may be performed if it is required.

4. Expand all types which are not primitive and repeat 1, 2, and 3 using the type operations specifications as requirements.

5. The process is complete when the desired level of implementability has been achieved.

6. Verify the performance requirements by analyzing the design.

4. TOOLS

GENERAL DISCUSSION

Goals and Purpose of Support Tools

The activities of software system design result in a collection of specifications and information about a system which provide a description to be used in a variety of ways. The system customer and system analyst studies this description to determine if the design will satisfy its intended requirements. The system designer uses the description to verify the correctness of the design and to analyze the performance of the resulting system. All levels of management use the design description to enforce procedures, to track development, to manage resources, and to develop status reports. Finally the system implementor uses the design description as a blueprint for constructing a concrete system out of existing materials such as hardware, operating systems, utilities, and programming languages. Thus the product of design activity is a collection of information describing a design to a wide variety of people.

The form of the design product is usually a set of documents which is tailored to fit the needs of the activities performed by the various development personnel. Because of the tedious and time consuming functions necessary to produce the design documents, they are often incomplete, out-of-date, wrong, or simply not produced. This results in the development of systems which do not perform their intended functions, or which do not meet performance constraints, and which usually contain many design and implementation errors.

Tools to support the software designer are intended to increase productivity by removing repetitive, error-prone clerical tasks. Tools which can be computer based (e.g., text editors, cataloguing systems) are less likely to require the designer's valuable time. The tools provide convenient interfaces to on-line design documents by supplying automatic analysis of the

design, editing capabilities, retrieval capabilities, and automatic report generation. Furthermore, certain clerical activities such as text entry, document retrieval, and editing may be performed by authorized persons other than the designers. For this reason computer-based tools are a valuable part of the methodology package.

Notational tools and organizational techniques are also valuable for many functions and lend structure to often unstructured activities. For example, a tool system may control access to the design documents by unauthorized people, thus improving the integrity of the design documentation. Such guidance reduces the possibility of mishap, though it does not eliminate it.

Useful tools are those which are flexible but limited in scope. A well defined tool which supports a specific technique will be successful if it is implemented properly and accepted by a wide audience.

The purpose of a tool system which supports a design methodology is to supply a computer based design information system that enhances the production and use of the design product. That is, all information relating to a design is represented as a data base management system whose operations are used by the development personnel in the derivation of the design, and in the production of a system. The operations of the tool system should support the procedures of the design methodology by performing the following functions:

- provide on-line access to the design information:
- provide interactive and convenient mechanisms for the creation, modification, and retrieval of the design information;
- provide mechanisms for describing and locating relationships between information objects;
- provide automatic analysis of the design to aid in design verification and static performance assessment;

- provide operations and support for automatic documentation retrieval;
- provide automatic status reporting and resource management;
- enforce design procedures;
- enhance management of the design process;
- guarantee the integrity and security of the design information; and,
- provide communications facilities for use by the development personnel.

Design Notations

The requirements of the methodology support tools depend upon the choice of notation used to represent the design product. The notation chosen for the RDM was discussed in the last chapter and is presented in detail in appendix A. It is based upon the notation of the WELLMADE methodology. This notation contains components for design documentation, specifications, and a program design language. No attempt was made to include a formal assertion language to supply specifications in the RDM. However, a formal syntax has been defined for all other notational components. The BNF specification of this syntax is given in appendix C.

Other choices of notation which have been used to represent the software system design components are charts, implementation languages, procedural and non-procedural design specification languages, and requirements specification languages. In the remainder of this subsection each of these alternatives are evaluated for use in the RDM.

Charts -- Charts are widely believed to aid the user in constructing a program with minimum omissions. The flow chart has been used in the traditional approach to software development. Its obvious strength is that it conveys the intuitive meaning of the detailed logic of a program, without requiring the reader to become familiar with a programming language. Its greatest weakness is the lack of discipline it allows the user. The number of ways in which flow charts are misused are as varied as the number of programs which have been described using the technique. For example, the contents of a box on a flow chart may contain programming language statements, or it may contain such abstract sketches of function as to be unreadable.

A second more recent form of chart used as a software development tool is the HIPO chart [IBM73]. The "Visual Table of Contents" portion is useful for managers to examine overall system construction. The "Overview Diagram" and "Detail Diagram" can be used for further explosion of system components. A drawback to the use of HIPO charts in particular, and all charts in general, is the vagueness of the semantic definition of the content.

An additional charting technique has been proposed by representing detailed program design through Schneiderman Charts [NASS73]. However, these charts also suffer from many of the same difficulties as all other charts.

Implementation Languages -- It is possible to use implementation languages to help communicate ideas between designers and specifiers. Much of the design work of the Multics system has been done in PL/I [ORGA72], a cumbersome yet powerful language. The use of PASCAL and its derivatives as a design language has recently become very popular [ICHB74, JENS75, KARA77]. PASCAL provides many of the constraints and attributes of structured programming which assist the designer in eliminating errors while producing well formed programs. MODULA has begun to be used for design activities involving real-time and concurrency [WIRT77]. More recently, the concepts of a programming language and design methodology have been combined causing the development of several new languages such as CLU [LISK74], Alphard [WULF76], and Euclid [LAMP77].

The major drawback of using any programming language to design programs is that the implementation is often tested before the design is complete. This violates the principles of representation independence and execution independence. It is extremely difficult to design in a programming language without considering implementation constraints.

Design Specification Languages -- Design specification languages or program design languages (PDL) are collections of statements with usually informal syntax rules used to describe the operation of a computer program. Certain of the languages are non-procedural. The SRI Special language, based on Parnas modules, describes systems in terms of their effects and structure [ROUB77]. It does not imply sequence or flow of control. These types of languages may be valuable tools in that they force the user to think in non-procedural, machine independent terms. However, they often are difficult to use if one visualizes a system design in terms of its flow.

P and V notation from the WELLMADE methodology combine a procedural notation for specifying program designs and a set of constructs to define data representations. The use of this notation is embedded in a structure which characterizes system designs in terms of modules containing programs and abstract data structures [HONE77b, HONE77a].

Requirements Specification Languages -- Languages in which to specify requirements present a new and very promising area of research. These tools could help designers talk to specifiers in rigorous, formal terms. It is believed that the majority of system design difficulties arise from improperly specified or misunderstood requirements.

The SADT system incorporates a requirements specification technique using functional architectures [ROSS77]. The architecture of a system is specified graphically and analyzed by a technical team. Another requirements technique, RSL, was developed for BMDATC and uses language primitives to decompose requirements functionally [DAVI77]. ISDOS uses a Problem Statement Language to describe a systems functional requirements and a

Problem Statement Analyzer to modify, add to, analyze, and generate reports based on the functional requirements of the system [TEIC77].

Design Personnel

The requirements of the methodology support tools must include a description of the functionality provided by the operations of the design information system. Thus, the various types of personnel which use the system must be identified by the type of operations which they perform on the system.

Five types of individuals have been identified as having major responsibility in any design project. However, these personnel types may be further subdivided or combined to fit the organizational needs of any development project. These categories have been chosen since they adapt naturally to most design projects and they fit the hierarchial organization of the design and the design procedures. The following is a brief description of the personnel types, their requirements, and their activities. (See Figure 8.)

Customers -- These individuals are external to the design process and initiate a new design project by generating a set of requirements. In addition, the customers are involved in all external reviews; they approve or disapprove of mission descriptions, they modify requirements, and they accept or reject the final design. The customers must be able to track the design process by receiving status reports and are generally given the access rights to read the mission descriptions.

Administrative Management -- They are responsible for initiating new design projects, allocating resources to the design project, planning and tracking the status of the project, planning and conducting external reviews, negotiating revisions of customer requirements, and submitting the final design for implementation.

Technical Management -- These individuals are responsible for all hierarchy management, module specifications through a requirements analysis,

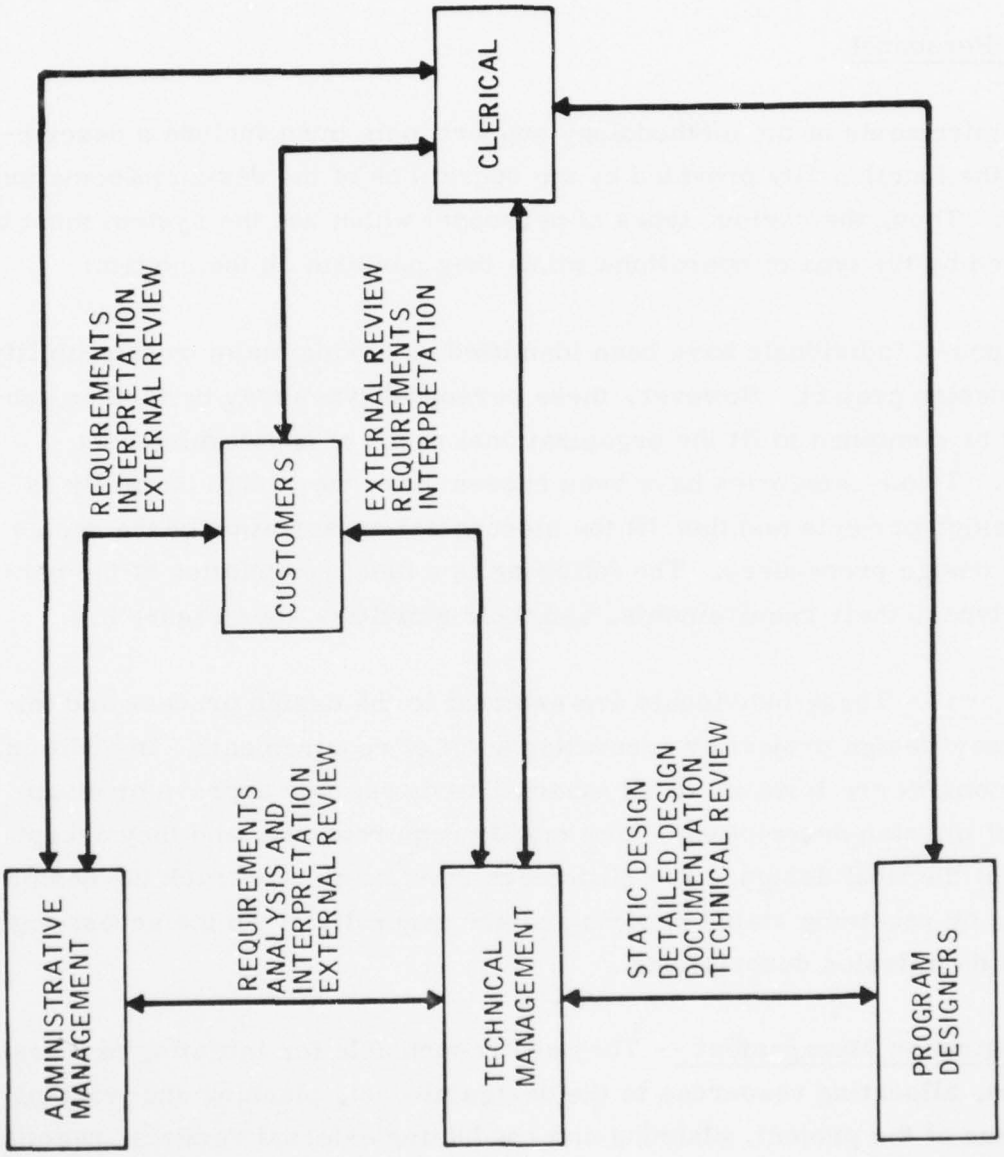


Figure 8. Personnel Interaction

requirements interpretation and static design activity. That is, the technical management performs system analysis activities. In addition, they are responsible for all preliminary designs, and therefore they will perform both static and procedural designs on the upper level modules. They assign programs and subsystem design responsibilities to program designers, they track and do status reporting on all design activities. Technical management conducts both internal and external reviews, authorizes changes, and freezes components of the design.

Program Designers -- The activities of the program designer are similar to those of technical management except for management responsibility and customer interfaces. The program designer is chiefly responsible for using the constructive approach for designing programs, the informal verification of the design, and performing static assessment of the program's performance.

Clerical -- The major activity of clerical personnel is data entry and retrieval. All authorizations to access data are given to the clerical personnel by the appropriate design personnel. The operations which are performed at this level are mainly text processing on various data types.

Structure of the Support Tools

The tool system will usually be a subsystem of a general purpose operating system. It will be a data base management system and data analysis system for managing and analyzing design information. Thus it can be specified as an abstract data structure representing the design information and whose operations specify the requirements for managing and analyzing that information.

The proposed tool system structure is similar to that of any general purpose operating system in which there is a human interface at the highest level and a resource management and hardware interface system at the lowest

level. We can identify at least three abstract levels of modules for the tool system. This logical structure is shown in Figure 9.

At level 1, there is a command and control system whose purpose is to provide the human interface and to enforce the access control mechanisms. A command language, similar to that of any interactive system, is supplied to the user for invoking permitted tool operations. The syntax for this language has not yet been specified, however of more importance is the discovery of the operations and the specifications of their requirements.

The command interpreters at level 2 are categorized by three general classes of operations to be supplied by the tool system. The general design tools are used for the creation, representation, and retrieval of the design information. The management support tools supply the aids for controlling information, activities, and resources used in the design process. Finally, the design analysis system aids the designer and system analysts in verification and performance assessment functions. Each of these categories will be discussed in the following three sections of this report.

The operating system interface supplies the data structures and operations necessary to implement the tool system in a given environment. Note that no specification of these facilities should be given until their requirements have been established by designing the structures and programs of the upper levels which use these facilities. However, as usual, a good designer can often anticipate the facilities to be used for implementation and will structure the requirements in a way such that they are satisfied by the specifications of the existing tools.

Any operating system which supplies facilities for convenient software development will provide a natural interface to the tool system. Such a system is the MULTICS system designed to support software development. Many of the tool system requirements are satisfied by MULTICS facilities for protection, data integrity, communications, information structures, information linking, text editors, run-off systems, etc. Note that the only tools which specifically depend upon the methodology are those relating to

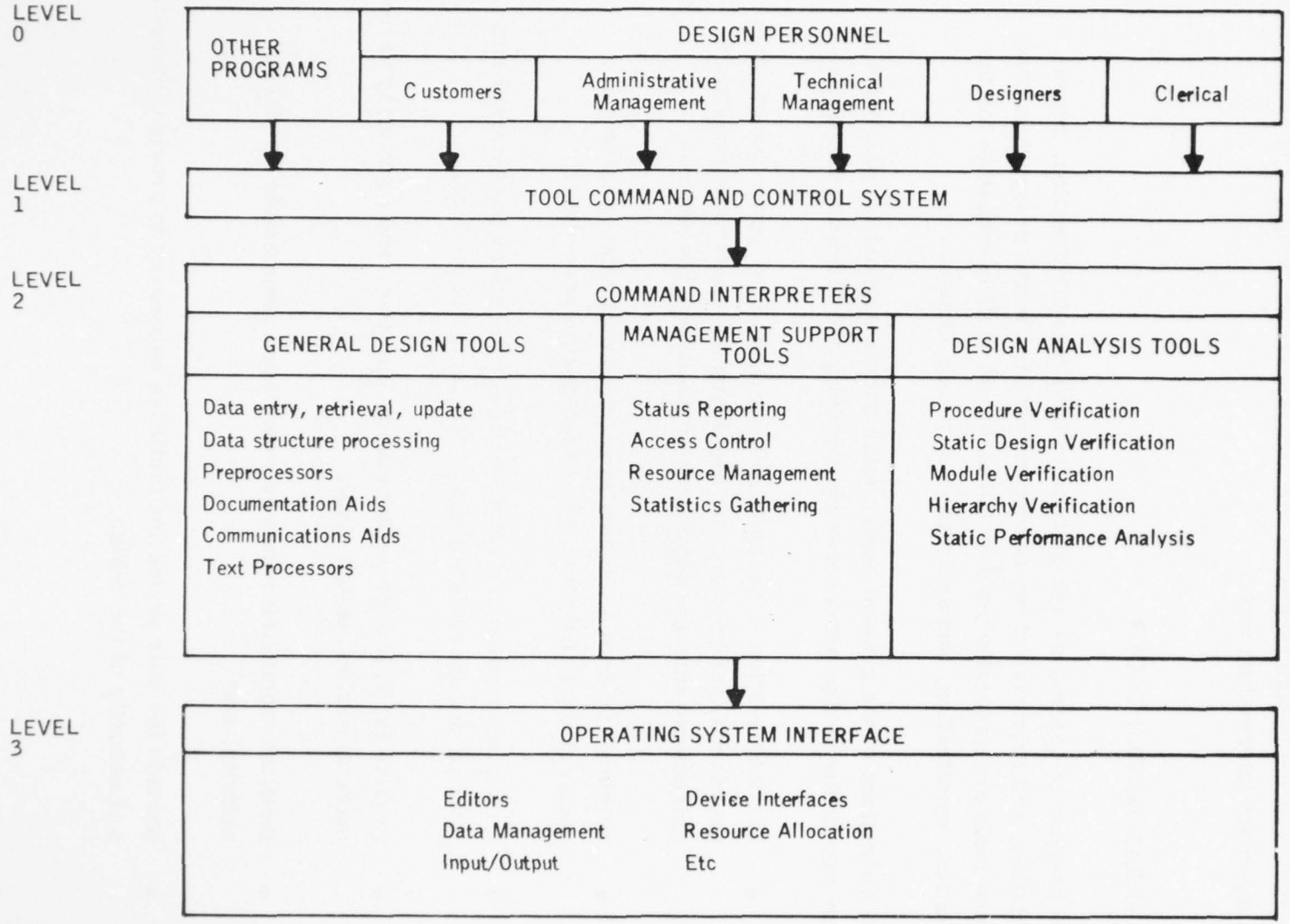


Figure 9. Structure of the Tool System

the syntax of the design language, the analysis of the design, and some management information tools.

GENERAL DESIGN TOOLS

The purpose of general design tools is to facilitate the creation, representation, and retrieval of the design information in the design data base. These tools are generally used by the technical staff responsible for the creation, verification, and analysis of a system design.

The operations of the general design tools should support the activities of each step of the design procedures by supplying the following facilities:

- on-line creation of design specifications at the static design step, procedural design step, the requirements analysis step, and requirements interpretation step of the design procedures,
- retrieval of design information at the appropriate design and requirements analysis steps of the design procedure,
- facilitate creation and retrieval of documents at the external and internal design review steps of the design procedure,
- provide facilities for backtracking, updating, and correcting errors made at earlier design steps,
- facilitate communications between those responsible for design activity, and
- provide the data protection facilities necessary to provide integrity and security of the design.

Data Types of the General Design Tools -- The data of the general design tools is a collection of information objects whose types are defined by a basic set of abstract data structures. To discover these data types we consider the hierarchical design and its documentation as prescribed by the program design language. Furthermore, we consider the information required to create new information at each step of the design procedures. Thus, using the type expressions of the program design language to describe data structures we arrive at a basic abstract data type for representing the data of the general design tools.

The design data base defines a collection of design projects. Each project is described by an identifier, a customer, a set of requirements, a project status report, and a design. The customer is described by an identifier and a set of customer generated requirements.

```
design data base:  project array

project:   (id:project_name,
            customer:cust_req,
            requirements: req_set,
            status:project_status,
            design:system)

cust_req: (id:cust_name, requirements:text)
```

The system design is specified as a hierarchy of machine designs. The machine design is described by a machine identifier, a definition clause which is used to map data types onto a system hierarchy, a mission description, a machine design, a collection of program designs, and a machine status report.

```
system:    machine array

machine:  (id:machine_name,
           definition: (type_name) array
           mission:  mission_desc,
```

```
design: machine_design,
program: (prog_design) array
status: machine_status)
```

A mission description is given by the three documents called the functional description, the usage information, and the acceptance criteria.

```
mission_desc: (functional: text,
               usage: text,
               acceptance: text)
```

The machine design specifies the environment, the state space, and a list of programs which operate on the machine's state space. The design is annotated with an overview and a notes document. The machine environment describes a set of requirements which must be satisfied by lower level machines. Requirements are given as a collection of requirement specifications on abstract data types. The machine's state space is specified by a collection of object declarations and a data invariant.

```
machine_design:  (overview: text,
                  environment: req_set,
                  data: data_desc,
                  prog_list: (prog_name) array,
                  notes: text)
req_set:  (type_desc) array
type_desc:    (id: type_name, requirement: v-notation)
data_desc:    ((id: object_name, type: type_desc) array,
                inv: assertion-notation)
```

The program design is described by the program identification, local variables defining its local state space, the program's static specification, the program's procedural design, and a program status report. Furthermore it is documented by an overview and set of notes.

```
prog_design: (id: prog_name,
              overview: text,
              variables: data_desc,
              prog_list: (prog_name) array,
              specif: (input: assertion-notation,
                       output: assertion-notation,
                       performance: assertion-notation),
              procedure: p-notation,
              notes: text,
              status: program_status)
```

Finally, we shall leave the data types for names, status reports, and the four basic information units – text, p-notation, v-notation, and assertion-notation as abstract data type designators.

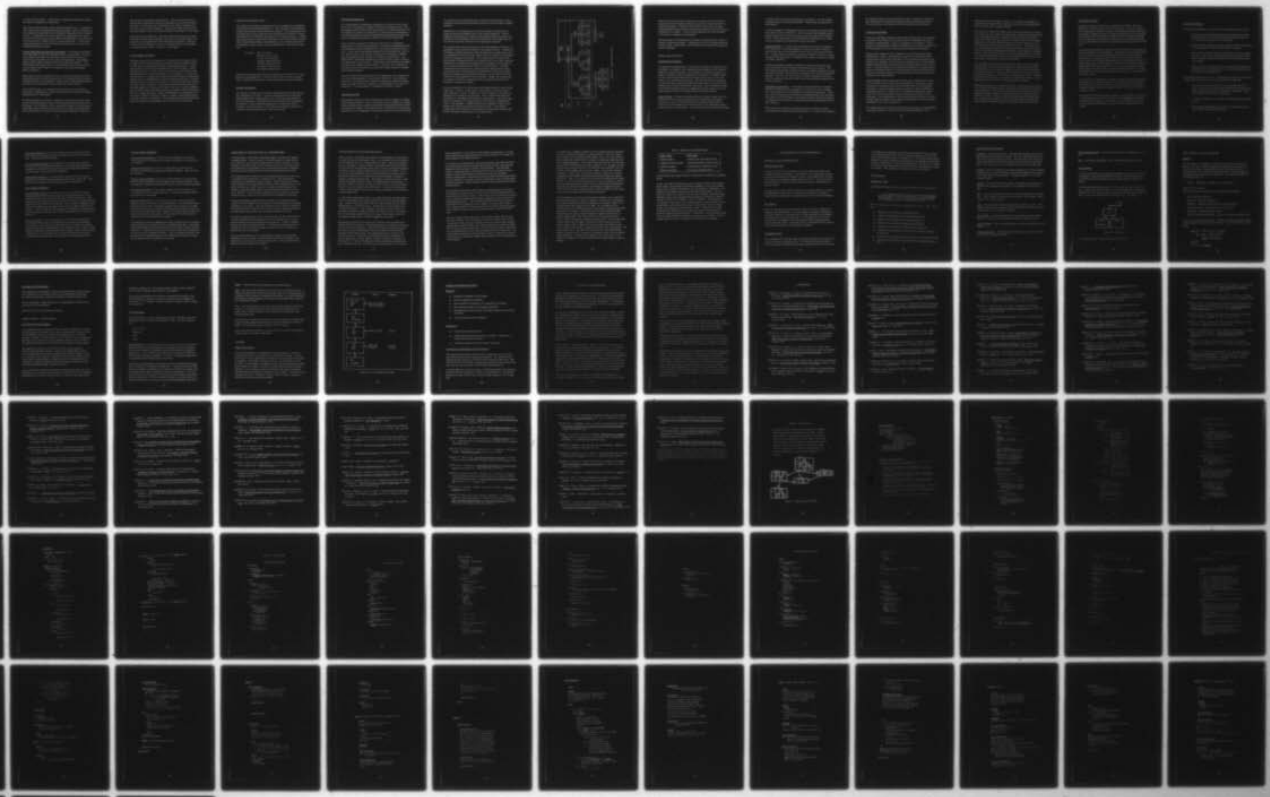
```
project_name, cust_name, module_name, prog_name: abstract
project_status, module_status, program_status: abstract
text: abstract
p-notation: abstract
v-notation: abstract
assertion-notation: abstract
```

Using this abstract data structure to represent the data type of the design data base the following observations can be made concerning the basic data types and their operations.

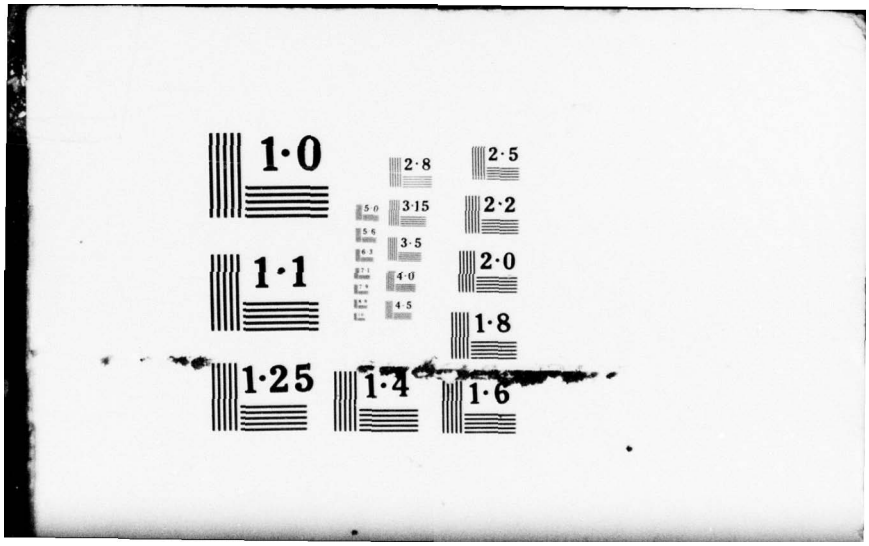
The two data structuring operations for data types, cartesian products (,), and array, represent relationships which exist between the various objects of the data structures. There is an implied set of operations for cartesian products and arrays which permit selection of objects and manipulations of the structures. Thus these structures and their operations may be implemented by a set of directories and directory operations for locating, retrieving, and creation of objects or they may be implemented by a relational data base system. Since all basic information objects are named the mechanism used for locating objects or defining relationships between objects may refer

AD-A062 404 HONEYWELL INC MINNEAPOLIS MN CORPORATE COMPUTER SCIE--ETC F/G 9/2
RATIONAL DESIGN METHODOLOGY.(U)
SEP 78 D BOYD, A PIZZARELLO, S C VESTAL F30602-77-C-0043
UNCLASSIFIED 47338 RADC-TR-78-208 NL

2 OF 2
ADA
062404



END
DATE
FILMED
3 -79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6
6·3

to these objects by name. In particular if a directory mechaism is chosen, the object is identified by its path name.

Each major design unit has a status report associated with it. The information in this report is intended for management usage. The status data structure contains the identification of the personnel responsible for the design of the unit. Furthermore, the access rights of the information units may be shared either with the user in the personnel identification as a capability list or in the name of the design unit as an access control list. In either case, a protection system can be constructed which provides the security of the individual information objects.

Clerical Operations and Interactive Processors -- The information objects whose data types are text are generally used for documentation. The creation, editing, formatting, and retrieval of these documents can be performed by any text editor with the usual set of operations. The Program Support Library (PSL) contains many of these operations [RADC]. The WELLMADE methodology utilizes a documentation organization and extraction tool called "doca" which is the designer's interface to the project library [HONE77b].

Most computer systems intended for use as software development facilities include file systems of sufficient generality for organizing software projects. The MULTICS file system includes a large number of organizational features and tools as do several DEC systems [ORGA72, DEC69, DEC76].

For all clerical tools, the important facts are that they are well human-engineered, comprehensive, suited to the particular environment and totally integrated into the methodology.

The three data types, p-notation, v-notation, and assertion-notation represent the formalized notation used to define analyzable specifications of a system design. Assertions are used in v-notation to define the requirements of a data type and in p-notation to characterize intermediate states, preconditions, postconditions, invariants, and termination conditions as well as for

data invariants and program specifications. Thus the processors for p-notation and v-notation will require the operations of an assertion-notation processor. At this time in the development of the RDM no definition has been given for an assertion language. Therefore the tools which treat the assertion notation as text and their processing is performed by a text editor.

Processors for p-notation and v-notation include preprocessors for shorthand input and local syntax checking, text-editors for making changes and corrections, and postprocessors for formatting and printing. Other processors for these notations are used in the design analysis and described in "Design Analysis Tools," Chapter 4, of this report.

General Support Operations

Additional design data base operations and facilities which would be useful to support tools for the general design activities include tracking facilities, version control, information retrieval, and documentation aids. A tracking mechanism would record all modifications, all verifications, and all backtracking by date to ensure that the most recently created and verified information objects are used in analysis or documentation. A version control mechanism is a facility which will archive early versions of information objects whenever modifications are made or backtracking occurs. Thus the designer is able to recall early versions of the design during design analysis or during redesign activities. Information retrieval aids should permit the automatic display of the design at any level. That is, a request to display a project should produce the display of all known modules in hierarchial order, whereas a request to display a module should display only information relevant to the module or a request to display a program should display only information relevant to the program. Documentation aids include facilities for document production, distribution lists, and various run-off formats.

DESIGN MANAGEMENT TOOLS

The purpose of the design management tools is to facilitate the management of all activities of the design procedures. The management responsibilities include both technical and administrative activities. The identification of the status data types of the design data base specification is intended to include all information to facilitate the management of the design process. A typical status data structure would contain information objects describing the design state, identification of design, responsibility, access control information, tracking data, version data, resources allocated data, and resources used data.

```
unit_status:  (state: unit_state,
               designer: person_name,
               access: access_control,
               tracking: modification_list,
               version: version_control,
               resc_alloc: resource_list,
               resc_used: resource_list)
```

Four areas of management have been identified as areas which may be supported by methodology tools. They are referred to as resource management, methodology management, information control, and data gathering.

Resource Management

The resources managed by the resource management tools include personnel, expenditures, and time. The operations provided by the tools will allow management to assign dollars, time, and design responsibility to given units of design. Tracking of these resources may be accomplished by automatic or semi-automatic updating of the design state and the resources used data structure. The design state identifies design, documentation, and verification states of the unit of design.

Methodology Management

The responsibility of methodology management is to insure that the procedures of the methodology are adhered to during the design process. Thus the management tools must provide operations for recording milestones, for controlling access rights to information, for ensuring that backtracking occurs when redesign is required by invalidating the appropriate design units, and by providing communication facilities for use by the design staff.

Tools to aid the management of software projects are not sufficient in themselves to insure successful projects. Techniques for applying the tools and procedures to guide development are also important pieces of the process. A total system management approach has been proposed by Turn, et. al., which is based on the mission requirements formulation [TURN76]. This method falls into the category of techniques and procedures. The WELLMADE methodology encompasses many of the same ideas of mission description and frequent reviews. Additionally, WELLMADE has provisions for many tools to support the procedures. Currently, some interfaces to automatic charting mechanisms are provided to enable managers to graphically examine progress.

The most widely discussed technique for the management of a software project is the chief programmer team concept [MILL72b]. This technique consists of a work organization supported by production program libraries. Since it contains many of the concepts of a methodology, it has come to be called one. It is more properly a management structure, however.

Information Control

Information control refers to the management of access rights and management of design versions. That is, operations must be available to the appropriate management for granting access rights to information objects and for removing access rights. This control can be used effectively to ensure that milestones of the methodology are met and to maintain the security of design.

The integrity of the design information is maintained when proper version control is used and when information is placed in archival storage at regular intervals.

Management of the design process requires available information about design activity and controlled access to all design information. Thus an additional requirement of the general design support tools is that there be a mechanism for controlling access to all components of the design data base and that this control be exercised by the appropriate personnel.

The ring structure used for protection in MULTICS provides a model for an appropriate access control mechanism of the design data base. In Figure 10, we display a possible hierarchy of data structures for a design project and a corresponding hierarchy of access rights and access control to be exercised by the five categories of personnel. At the highest level of control is the administrative management with access rights to any level of data structures. Furthermore, administrative management may change the access rights of any lower level personnel to any lower level data structure. Technical management has access rights to module directory information and structures at lower levels with appropriate capabilities to change access rights. Similarly, designers and clerical personnel have access rights and access control to the data applicable to their activities. Finally, the customer has access only to customer requirements and the mission descriptions and no ability to change access rights.

Access to instances of the various data types depends upon the nature of data of that type. However, in general the access rights of a data type will include operations to create, to destroy, to read, to update, or to execute an operation on an instance of the data type. Access control will involve the permission of a personnel type to perform an operation on a given data type. Thus the management of a design process can be aided by controlling this access. For example, a module which has been designed, verified, documented, and reviewed may be given only read access. The customer requirements may be frozen to avoid changes in requirements without negotiations and proper backtracking in the design process.

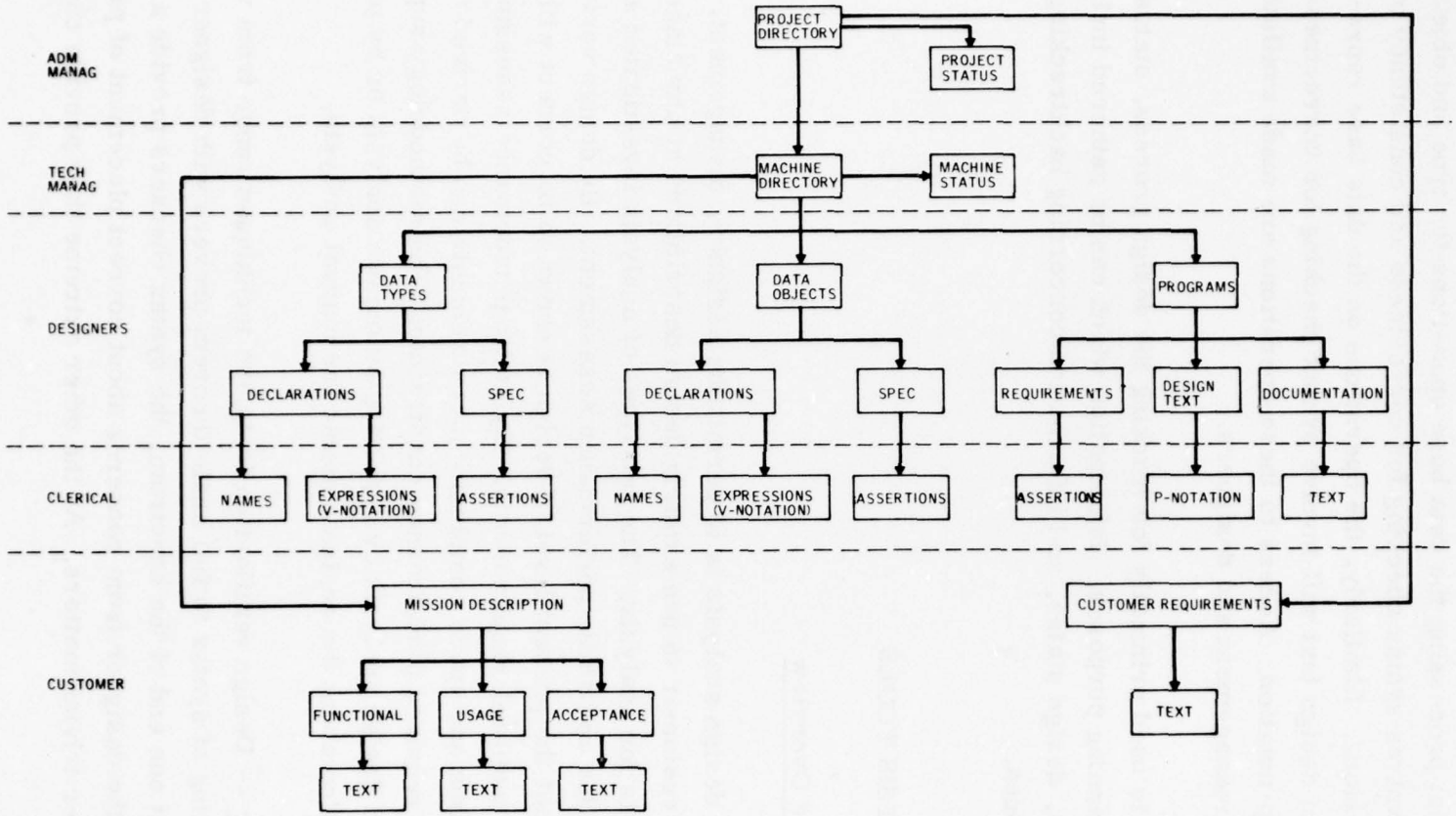


Figure 10. Access Control Levels

Finally, note that processing the data base operations for type and object declarations involves syntax checking for correctness and consistency of v-notation expressions. Similarly, the operations on the data base representing program design text will involve syntax checking for correctness and consistency of p-notation. Access to these operations are made available to all levels of management and designers.

Data gathering is used primarily for tracking the design process, status reporting, and planning purposes. Information which can be gathered includes resource usage, design states, and information concerning backtracking and missed milestones.

DESIGN ANALYSIS TOOLS

Introduction and Overview

The purpose of design analysis is to permit the designer, management, system analyst or customer to use existing design descriptions at any stage of design as a basis for analysis. The two types of analysis investigated are design verification and static performance assessment. By design verification we mean that the present level of design is shown to be correct with respect to its functional requirements. By static performance assessment we mean that time and space complexity measurements can be derived and compared with required performance constraints. The methodology support tools should aid design analysis by supplying automatic analysis or by providing information about the design suitable for manual analysis.

Syntax Checker -- Design verification tools and techniques range from very informal checking of syntax to full scale theorem provers with designer interaction. At one end of the spectrum, the syntax checkers provide a means to free the designer from concerns about correct placement of punctuation and other trivial matters. At the other extreme they provide checking

of global syntax such as type and hierarchy consistency. The SRI methodology has syntax checking tools [ROUB77]. Similar tools are planned for WELLMADE.

A typical example of a global syntax checker is the Design Assertion Consistency Checker (DACC) [BOEH75]. These automated tools typically search program designs for interface attributes such as parameters and returned values. These elements are then matched with the definition of the called routine and display any inconsistencies which are found.

Semantic Checker -- A second type of verification aids may be termed formal verification. The current approach to theorem proving is related to past attempts (artificial intelligence) but has taken a more pragmatic direction. A large degree of designer interaction is allowed in the more successful theorem proving systems. Examples of these types of systems are found in [YEH77, WEGB77].

Predicate transformer applied to program verification involves the selection of invariants and the statement of assertions about a program. The WELLMADE methodology is based entirely on the Dijkstra guarded command constructs and logic, and therefore is well suited to the assertion generation approach. Similarly, the work of Katz and Manna in proving correctness uses the analysis of invariants which are semi-automatically generated from the text [KATZ75].

Performance Assessment -- The analysis of the performance of computer programs, implementations or designs is included in the area of verification, since performance characteristics are actually requirements. There are very few notational tools available for specifying performance attributes and criteria of system designs. One notable attempt to analyze performance of a design was the Design and Evaluation System (DES) developed by Honeywell [GRAH73].

DES successively used more detailed versions of the design to predict increasingly fine-grained data about performance. The information displayed

for analysis included a list of all procedure calls, an estimate of space and time requirements, indications of progress and any inconsistencies found. The second phase of the analysis involved simulation of the procedures.

Verification and Design

It has been repeatedly emphasized in this report that the basis for the design procedures of the RDM are the concepts used for verifying that the design satisfies its specifications. That is, the approach is constructive - the refinement of specifications of software requirements, the software design, and the construction of its correctness proof must all proceed in parallel.

Design activity begins with the recognition of a set of abstract functional specifications. That is, specifications are given as a set of input/output assertions and data invariant assertions on the underlying state space of the program to be designed. The state space is defined by the set of all possible program data structures and the range of permissible values of these structures. A correct program is a mechanism which defines a transformation in which the mechanism starts in a state satisfying the input assertion and then halts in a state which satisfies the output assertion and which maintains the assertions of the data invariant.

Using knowledge of the program specifications and a notation whose semantics are defined by their transformational properties, the program text can be derived by considering only correct transformations on the underlying state space. Thus the program designer constructs a correct program by constructing its proof. Furthermore, the designer identifies new requirements in the form of operations acting on abstract data structures in the state space. The requirements of these operations must be given in the form of input/output assertions on the underlying state space in order to maintain provability of the current design level.

The design procedure of the RDM requires that the designer be knowledgeable of the proof technique as well as the transformational properties of the

notation used for specification of the text. By using this knowledge it is argued that the designer will be capable of constructing and specifying the text of provably correct program designs.

The tools which support the design process and which enhance the verification activity are those which force the designer to follow the procedures and to use the notation correctly. Examples of these tools are those which require that input/output specification are included as part of the program design text, or that each abstract operation used in the program text be declared and that input/output specifications be given. Tools which check for correct syntactic usage of the notation and which report inconsistencies such as type violations and missing declarations, also enhance verification activity. Hierarchial consistency can be tested automatically by ensuring that each abstract data type and each abstract operation be defined by data declarations and programs in lower level modules.

Note that all tools mentioned above fall into the class of extensive syntax checking tools. That is, the PDL includes all components of the design document which are necessary to support and to force certain activities of the design procedures. Extensive syntax checking can be used to ensure consistent and correct use of the PDL for design documentation. By enforcing certain syntactic rules, the designer is reminded of activities and conventions which must be used to ensure provability of design.

Specification languages, assertion generators, and theorem provers necessary to perform automatic proof of correctness are beyond the scope of current investigation. The realization of a practical theorem prover which is suitable for program correctness proofs will require a major amount of additional research. However, tools which may be called semantic aids, such as specification languages and assertion generators based upon the design language semantics, appear to be feasible within the near future.

Verification Technique

Two types of design verification are considered in the RDM. They are identified as technical and non-technical verification. Technical verification is supported by formalized notation and rigorous procedures which allow the designer to demonstrate that the design specifications satisfy the design requirements and that the program design satisfies the program specifications. In principle, all technical verification can be handled automatically or semi-automatically provided mathematically rigorous notations have been used.

Non-technical verification is strictly manual and involves the inspection and review procedures of the methodology. The purpose of non-technical verification is to ensure that the formal description of requirements, design specifications, and program specifications satisfy the intended customer requirements for the programming problem or system design problem. The support tools which aid non-technical verification are those which supply the appropriate design documentation for review and which guarantee that the information in these documents has been technically verified at the current level of design. Tools for documentation and tracking have been described in earlier sections.

In the absence of a notation for assertions, automatic verification of semantic correctness is impossible; however, the tools can supply the design information necessary at each step for manual verification. Furthermore, automatic theorem provers and semantic analysis are beyond the scope of current research.

All syntactic verification and type checking can be automated in cases where a formal syntax has been used. In the present methodology, this includes p-notation and v-notation (see Appendix A). In the following section the areas of technical verification are outlined and the support tools are discussed.

Technical Verification

The technical verification of a design has been subdivided into four areas.

1. Hierarchical design verification involves verifying that the design is represented by a well-formed hierarchy of machines and that it is complete with respect to its requirements.

2. Machine design verification involves verifying that the module design is well-formed and is correct with respect to its syntax.

3. Static design verification involves verifying that the static specifications of a machine (state space and program specifications) consistently satisfy the requirements which have been derived from high level modules.

4. Program design verification involves verification that a program is well-formed, is syntactically correct, and is semantically correct with respect to its specifications.

The following definitions of terms are applicable to the specific checks which must be made for each of the four areas of technical verification.

1. An abstract data type specification is a named data type whose values are described by a type expression defining an abstract state space and whose operations are specified by input/output requirements defined over the abstract state space.

2. A machine environment is a collection of abstract data type specifications.

3. The functional requirements of a system are specified as one or more abstract data type specifications.

4. A type expression describes the structure of objects of a data type. The structure operators are ';' (ordered cartesian product), ',' (unordered cartesian product), ' ▌ ' (discriminate union) and array.

5. A subtype of a data type is the type of any of the objects of its type expression.

6. A type is primitive if

   a. it is defined by the PDL of the methodology, and

   b. it has no specified operations which are not primitive (defined by the PDL of the methodology), and

   c. all of its subtypes are primitive.

7. A type is defined if

   a. it is primitive, or

   b. it appears in the DEFINE clause of a machine specification, or

   c. each of its subtypes are defined and it has no operations which are not primitive.

8. A machine hierarchy is a partial ordering of machines in which all machines whose environments contain data type specifications for an undefined type precede the machine in which the type is defined.

The checking activities for each of the four areas of technical design verification are now outlined.

Hierarchical Design Verification

Environmental Consistency -- If the environment of two or more machines requires the use of the same type, then it must be shown that the specification of requirements are consistent. The tool system can only display each set of specifications and allow the designer to visually check

consistency. Thus, whenever a new data type is entered into the environment of a machine design, the tool system can automatically locate all other occurrences of that data type, print each requirement, and permit the designer to make this check.

Hierarchical Ordering -- The hierarchical ordering of a system is well formed when all environmental consistency checks have been made and all static design verifications have been made on existing machines. The hierarchical ordering can be checked automatically. That is, when a new machine is designed, it defines one or more data types. The new machine appears at a lower level in the machine hierarchy than those which use the data types it defines. The tool system can automatically locate all machines which use the defined data types and request environmental consistency checks and static design verifications. Secondly, if a type which has been defined by a machine design is later used in the environment of a new machine, that new machine can be automatically inserted into the approach hierarchial position and the environmental consistency and static design verifications requested.

System Closure -- A system design is complete when all undefined types of a system's functional requirements and all module environments are defined by some module and the module has been verified. The system closure check can be automatic by maintaining a list of undefined types and by tracking verification activities. Note that this is only a syntactic check which ensures that all non-primitive types are defined by a machine design.

Machine Design Verification

Data Type Syntax -- The type definitions of the machines environment must be checked for correct syntax. By defining a v-notation parser this syntax can be checked automatically. Note that the requirements of data type operations are assertions which are currently treated as comments in the PDL.

Data Type Consistency -- All object data types of a machine and its programs which are not primitive must be declared in the machines environment. This check can be automated.

Machine Program Closure -- All operations of the data type defined by the machine and all subprograms internal to a machine must correspond to a program specification and program design (p-notation text) within the machine design. This check can be automated.

Global Data Consistency -- All global data variables declared within the programs of a machine must be consistent with respect to data type. This check can be automated by maintaining a machine symbol table.

Static Design Verification

Representation Validity -- The designer must verify that there exists a mapping between the abstract state space of the requirements for a data type and the state space of the machine which defines the data type. This mapping must be consistent by preserving the invariance on the abstract data space by the data invariant of the module. Since no formal mapping has been defined in the RDM, this consistency check will be visually verified by the designer. Thus the tools can only display the requirements of each environment containing the data type and the static specification of the module which defines it.

Operation Validity -- After representation validity has been established for an abstract data type, the designer must show that the input/output requirements of each operation of the data type are satisfied by the input/output specifications of the programs which implement them. Again, by displaying requirements and specifications the designer can prove that the input specifications are not stronger than the input requirements and that the resulting output specifications are not weaker than output requirements.

Program Design Verification

Program Data Consistency -- It must be shown that there are no undeclared objects used in a program text design (p-notation). This check can be automated.

Subprogram Consistency -- It must be shown that no undeclared programs are activated in a program text design (p-notation). This check can be automated.

Program Type Consistency -- It must be shown that there is no inconsistent use of type operations within a program text design (p-notation). This corresponds to strong type checking performed by several modern compilers.

Program Syntax Check -- The program design must be checked for correct syntactic use of the design language. By defining a p-notation parser, this check can be automated.

Program Semantic Check -- The designer must prove that the program design is correct with respect to its specifications. The development of this proof is the basis for the constructive approach and is central to the RDM. The proof is constructed in parallel with the design as discussed in the previous chapter on procedures and in Chapter 4's "Design Analysis Tools" discussion of verification and design. The mechanisms for formal proof of correctness are not discussed in this report.

The design of the tools for supporting the above fourteen checks of technical design verification will involve the use of additional data types. These data types will generally represent symbol tables which define symbols and their attributes. In addition, the status data structures of each unit of design can be used to track the validation activities. Thus, discrepancies between validation dates and modification dates can be detected and reported to ensure that the proper order of validation has occurred.

Design Order vs. Data Entry Order vs. Verification Order

The procedures of the design methodology support a constructive approach to design activity. That is, there is a prescribed order of design activity which should be used in system design and that order is top-down. Thus a definition of top-down design is that a program can not be functionally specified until all of its requirements have been identified and that the program design can not be specified until its functional specifications are known.

The management and enforcement of methodology procedures can only occur if activities can be observed in a systematic way. The tools provide a mechanism for tracking the design activities. Thus the normal procedure would require that the design data be entered into the design data base in the order in which it is created. However, because of backtracking and the existence of parallel design efforts, this top-down order of data entry may not be observed. Thus any order of data entry will be allowed to create the design data base, however, the verification order should occur top-down.

Top-down verification implies that the machines are verified in the hierarchical order of machines of the designed system beginning with the highest machine in the hierarchy. The checking order for each machine begins with hierarchical design verification, then machine design verification, followed by static design verification, and then program design verification.

Note that the closure checks for the machine design and hierarchical design cannot be completed until all programs have been specified for a machine and all machines have been specified for a hierarchy. Partially completed designs can only be partially checked. A requirement of all automatic verification tools is that checking continue even after errors or missing components have been encountered.

Errors and discrepancies are recorded and the designer is notified with appropriate error messages. Backtracking must occur to a position in the top-down hierarchy where corrections can be made. This requires that all affected machine designs be re-verified.

RECOMMENDATIONS FOR IMPLEMENTATION

In this section we shall make some general recommendations of alternatives for implementing the RDM support tools. Our discussion up to this chapter's section contains only a set of requirements for tools. Before any implementation is attempted, a system designer must design and document specifications for the tool command and central system and the command interpreters. The design activity would naturally follow the procedures of RDM. The resulting design/documentation would contain the functional specifications and procedural design for each command of the general design, management support, and design analysis tools. The designer could then choose an existing general purpose computer system for implementation of the design. If given the freedom of choice, the system chosen might be governed by implementation cost and predicted tool performance. A system which provides facilities closely matched with the tool design specifications would minimize implementation cost.

As indicated throughout this chapter, the requirements for the RDM support tools are based upon many of the requirements of the WELLMADE support tools. Several of the WELLMADE tools are under development and will be implemented on a MULTICS system. Furthermore, we have noted in the "General Discussion" section of chapter 4 that the MULTICS system was specifically designed to support software development activity. Therefore, all of the operating support functions and many of the general design and management support tools are already supplied by the system.

Specifically, all objects of the tool's data structures may be implemented through MULTICS segments. The data objects may be assigned to entire segment or archives of a segment. The MULTICS directories and associated operators are used to link together these data segments and provide the operations for locating objects, for creating new objects, and for access control to objects. Text editors are provided with macro capabilities to enhance interactive creation, updates, addition, retrieval, and inquire to data segments. The runoff facility allows for automatic formatting and display of design documents. MULTICS permits command files to be created,

thereby allowing for constructions of executable tool procedures. A memo system permits communications between various members of a design team by providing on-line mailing facilities.

To complete the implementation of the general design tools under MULTICS, the data structures of the tool design specifications must be organized and assigned to data segments, archives, and directories. The text editor macros, runoff procedures, and command files must be created to correspond to the procedural design specifications of each tool operation. Finally, preprocessors which permit short-hand input and local syntax checking of p-notation and v-notation must be implemented using an existing implementation language. Similar preprocessors are being created for the WELL-MADE notation.

The MULTICS facility which will enhance the implementation of several of the management support tools is the protection mechanism provided by the ring structure for access control. Data objects can be protected from illegal access, thereby permitting documentation management and enforcement of methodology procedures. The subsystem QUIKNIS which exists under the Honeywell GCOS III operating system supplies an on-line PERT system which could be used for resource management and status reporting functions. This subsystem could be converted to operate under a MULTICS environment.

All of the design analysis operations would require implementation using MULTICS facilities. The environmental consistency check, static design verification, and the program semantic checks are all based upon informal specifications and thus involve only display operations for manual checking. Just as with the general design tools, the MULTICS text editors runoff facility, and command files can be used for implementing these operations. All other analysis operations involving syntax, closure, consistency, and hierarchial structure checks will require implementation in an existing programming language.

If the RDM user is willing to modify the recommended RDM documentation structure, specification notation, or design notation, then a few existing tools could be used to implement RDM support tools. Note that certain of these modifications would involve a change in methodology reflecting a different design philosophy. We therefore recommend that the user proceed with caution if modifications are being considered. For example, by modifying the documentation structure, the PSL/PSA tools of ISDOS [TEIC77] could be used to supply entry, retrieval, update, and document analysis capability. However, this system was designed to support the creation, management, and analysis of requirements document rather than design documents. That is, no facilities are provided for handling formal specifications. It is recommended that this tool be used for producing a requirement interpretation document rather than the entire design specifications. Similarly, the tools for the SREM of the Software Development System [DAVI77] or the System Analysis Tools of Softech [ROSS77] could be used to generate the requirements interpretation document.

SRI International has designed a specification language, Special [ROUB77] for constructing module, machine, mapping, and hierarchy specifications. Each module specification of this methodology corresponds roughly to the specification of type requirements in RDM. By substituting Special specifications for the RDM type requirements and by reorganizing the RDM hierarchy of machines to correspond to the hierarchy proposed by SRI International, the analysis tools developed to support Special could be used to support RDM. The Special tools have been implemented for a PDP 10 operating under the TENEX operating system. They include some general design tools to support creation of the specifications and some design analysis tools to support hierarchy and type requirement checks. These involve syntax checks, strong type checking in the specifications, and hierarchial checks. The Special tools only involve static specification; program designs are not yet known to be included as part of the formalized design document. The four major handlers of the Special tools correspond very roughly to the RDM requirements for design analysis checks as follows.

Table 5. Special Tools and RDM Checks

Special Tool	RDM Check
Module Handler	Machine Data Type Syntax Check
Mapping Function Handler	Data Representation Validity Check
Interface Handler	Environmental Consistency Check
Hierarchy Handler	Hierarchial Ordering Check

All other RDM checks apply directly or indirectly to the analysis of program design.

Finally, note that if the RDM user were to replace the PDL with an appropriate subset of an existing implementation language, many of the program design checks would be performed by the early phases of compilation. The language should support abstract data types and strong typing. The compiler can be used for consistency checks, type checking, and syntax checks. Examples of this type of implementation language are MODULA [WIRT77], Euclid [LAMP77], LIS [ICHB74], and CLU [LISK75]. The difficulty of this approach is that most of the existing compilers require all levels of the design before such checks can be made. This prevents the incremental design and checking which is desirable in RDM. To provide this facility would require modification of the compiler to permit syntax checks on partially completed program designs. Furthermore, code generation should not occur until all design correctness issues have been resolved. That is, testing cannot be used as a substitute for verification of the design.

5. DEMONSTRATION OF THE METHODOLOGY

PURPOSE OF THE DEMONSTRATION

Statement of the Task

In order to test the ideas developed in the earlier tasks of the RDM work, a software design was produced. This design was to be performed using the procedures and techniques developed in Task 2, with support from the tools (or at least their anticipated support) defined in Task 3. The documentation of this design was to reflect the documentation structure developed in the notation of RDM.

In order to give a more realistic feeling of the possible problems of the practical application of RDM, the demonstration was produced by designers freshly trained and therefore, not experienced in the use of the methodology.

The Audience

There are two specific groups of people to whom the design is addressed: designers and implementors. Since the purpose of a design methodology is to aid the production of reliable software, understandability of the design to the implementors was very important. It is the experience of the demonstration team that designers may understand complex design procedures more readily than coders. Given that premise, significant effort was placed on clarity of design from an implementors viewpoint.

Anticipated Results

The assumption that a design could be produced using RDM in its purest (and most recent) form is obviously naive. The demonstration had the goal to test the teachability of RDM to good average designers together with the goal

of debugging the methodology and unifying its conventions and procedures. It was anticipated therefore, that the resulting design methodology might look somewhat different than its original form. The actual software design produced in the demonstration is of interest only for highlights and exemplifying the methodology. An interesting byproduct is, of course, the comparison of two designs following the same requirements, but produced by different methods. This comparison is discussed later in this chapter.

THE PROBLEM

Definition of a PSL

A programming support library as defined in RADC-TR-74-300, Volume V is

> ". . . the repository for data necessary for the orderly development of computer programs using top-down structured programming technology. . . A PSL also includes the necessary computer and office procedures for manipulating that data."

There is a wide variety of functions available to the user of a PSL. These are:

- Storage and maintenance of programming data
- Output of programming data and related control data
- Support of the compilation and testing of programs
- Support of the generation of program documentation
- Generation and maintenance support for PSL hierarchy
- Collection and reporting of management data related to program development
- Control over the integrity and security of the data stored in the PSL
- Separation of the clerical activity related to the programming process

Relevant Terms and Concepts

Libraries -- The data in the PSL is organized into internal (machine readable) and external (notebooks) libraries. Data in a single library usually shares some common origin or testing. This use of the word "library" is general and should not be confused with the more specific use of the word below. The hierarchy of data is organized into four levels as described.

Project -- This is the highest level of data organization in a PSL hierarchy. It corresponds to a major independent programming operation and contains all the data related to the operation. An example might be: "the Jovial compiler project."

Library -- The next level below a project corresponds to a particular version or major component of a project. For example: "1976 Jovial compiler."

File -- Libraries contain files, which are unique identifications of similar data. A file contains the same type of information. For example: "source code for 1977 Jovial compiler."

Unit -- The lowest level of data organization in the PSL is a unit. A file contains units which are segments of data such as lines of source code, an object module, or a document in text form.

PSL Catalog -- A directory of all PSL's which exist in a given computer system. A directory of all management data files is kept here also.

Project Catalog -- This is a data file used to record each library file in the project.

Library File Index -- This index gives the location of each unit in the file and of the file accounting record.

File Accounting Record -- An area used to store the accounting data for each file.

Key -- A character string (like a name) which identifies a unit in a file.

Purge Function

A number of functions are available to be performed on each entity in a PSL. For example, functions to add and delete projects, move and purge files. In particular, this demonstration used the purge function for units for its design.

The purge function allows the user of a PSL to remove a unit of data from a file. In order to remove this unit, it must be verified to not be included in another unit. (The inclusion concept is a way of textually incorporating data from one unit into another without repeating the data. See Figure 11.)

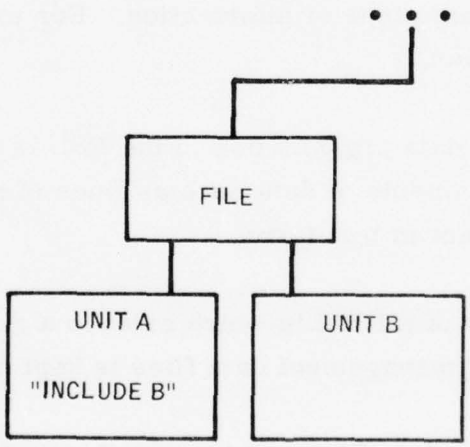


Figure 11. Data Shift

B is included within A and therefore cannot be deleted.

FIRST ATTEMPT – DATA DEFINITIONS

Approach

The first effort at using RDM to redesign the purge function involved the data of a PSL, only. After thoroughly understanding the requirements of the PSL, the designers examined all elements of the data structures it incorporated. This meant that the first step was to follow the input data through the system to the desired conclusion. The input data provided to the purge function was:

PURGE PROJECT, LIBRARY, FILE, UNIT, KEY

where KEY was optional.

This effort led to the uncovering of the following data structures:

PSL: collection of projects

PROJECT: collection of libraries

LIBRARY: collection of files and file accounting records

FILE: collection of units and unit accounting records

UNIT/KEY: collection of (source) data

UNIT ACCOUNTING RECORDS: collection of information about a unit.

Given the data structure described, it should be possible to extract the operations available for each type mentioned above. That reason led to the definition:

```
type file   (unit: T1, uar: T2) array
     fun    purge   (unit_name: T3)
            returns (success_fail)
end file
T1, T2, T3: abstract
```

Viewing the system in this way requires that much information about higher levels of the design be developed at the bottom level. For example, are pointers and unit names identical? Is the unit name a member of the array of file, or is part of the contents of unit abstract?

Advantages and Disadvantages

Given the experience level of the designers, the questions posed above presented a difficult situation. We found that we were able to traverse to the bottom of the system design for each particular function and attribute it to a known type. We were not then able to interrelate operations on types to each other, so that the design was complete (yet skeletal) from the top down.

The obvious advantage of this approach when isolated from other activities is then: the data structures and interrelationships of data were uncovered readily.

Conversely, the disadvantage is: the interrelationships of operations (or flow of control) was hidden until the end of the design process.

This disadvantage meant that environmental criteria such as execution speed, machine type, and implementability of the design were ignored.

SECOND ATTEMPT – ALGORITHM DEVELOPMENT

Approach

The success of the previous attempt was obviously limited. Therefore, the designers took a respite from data descriptions and began looking at the problem algorithmically. This approach embodied the standard top down structured programming approach with the addition of the use of the constructive approach.

This attempt yielded a very clear, diagrammatical understanding of the flow of control. In particular, the design resulting from this looked as follows in Figure 12:

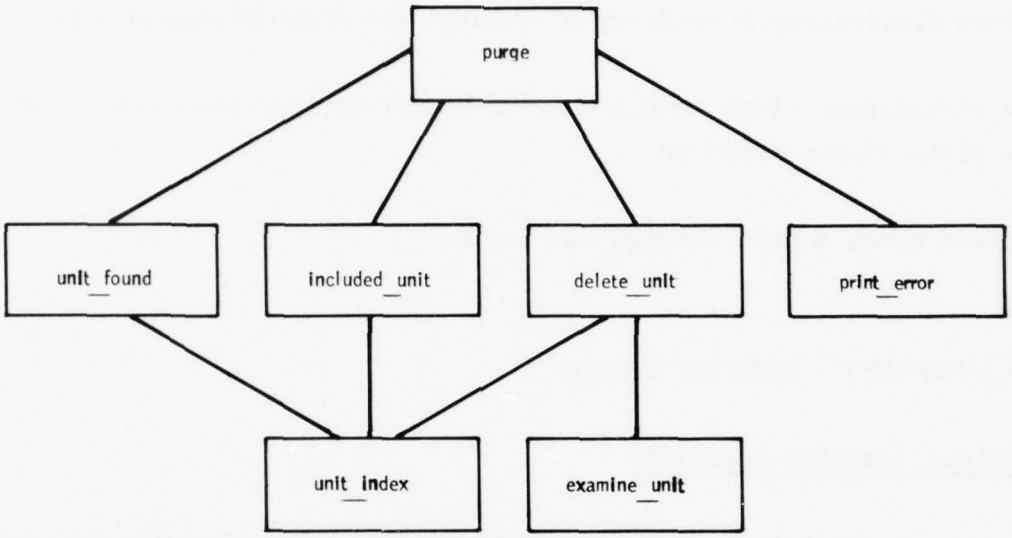


Figure 12. Program Structure Flow Chart

Without spending an excessive amount of time explaining an incomplete design, it should be noted that of the two lowest level functions, unit_index and print_error lead one to believe that the design rests on the location and referencing of a unit in a file. At this point the designers were tempted to introduce the selection of pointers, an implementation sensitive idea and deliver the design. Each function was derived using the constructive approach and therefore, felt to be correct. The control flow from module to module was clean, and preserved the idea of transformations on the input space to derive the output space.

Advantages and Disadvantages

The consultants to the designers felt the relationship of data items and types to the operations as prescribed by RDM was not sufficiently understood. The major disadvantages: tendency to overlook interrelationship of data.

And the advantages: a high conformance to implementation constraints and an ease of use of the technique.

With this in mind, a third attempt was begun.

THIRD ATTEMPT – SYSTEM DESIGN

The Abstract Machine Approach

It was suggested that the abstract machine concept as introduced in the previous description of the RDM in this report, might provide a means of unifying the algorithms and data structures developed from each of the first attempts. Briefly, an abstract machine may be viewed as a set of types and operations for those types for which application programs may be written. The application programs in this context are the programs being designed.

This approach in practice is an iteration between the data definition method and the algorithm method. A first try at defining data is made. Next, the algorithms necessary to manipulate that data are defined. Then the consistency of the system is examined and the process repeated as necessary until a machine design is completed. After a sufficient number of iterations, the logical groupings of type should lead the designer to a few, well-defined machines which comprise the system.

In the two previous attempts the designers had tried to develop in one case a hierarchy of data and in the other a hierarchy of algorithms; in the suggested RDM approach each and every step creates a machine as data types and as

programs running on it. The machine design is almost totally completed before further machines in the hierarchy are designed.

This approach is difficult to use without a background of familiarity with abstract machines, however, and may be rejected by some designers as too difficult. The "Critique" section of Chapter 5 contains a further critique of the approach.

The Final Design

The final design is a system containing four abstract machines, one corresponding to each of the levels of hierarchy in a PSL. The four machines were:

PSL_SYSTEM

PROJECT

LIB

FILE

PSL-SYSTEM -- This machine contains two major programs: Batch_control program and Online_control_program (not designed in the demonstration). Each of these programs is designed to process requests from a different input source. It is a reflection of our desire to stay within the IBM framework that this division was retained. We can see no design or implementation constraint which would require its retention.

The Batch_control_program was designed to show its relationship to purge, which it involves as a function on the type PSL. Note that all low level functions are designed as operations on the highest level type in the system PSL. This result evolved after a number of iterations in the design. The major reason for this was the fact that the functions and their data are also visible from the highest level. That is, the user requests a unit (low level) to be purged by submitting his request to the Batch_control_program (high level).

PROJ -- This machine was not designed in the demonstration.

LIB -- The LIB machine defines the type library as a collection of files. It contains a program (as a minimum) to purge a file unit. The purge_file_unit program in turn references a subprogram file_found. The major reason for the existence of these programs is to locate the particular file in which the unit to be purged resides.

FILE - The FILE machine is the most detailed machine design in the demonstration. It contains the type unit and the programs PURGE_UNIT and ADD1_UNIT. These programs in turn reference the subprograms unit_found and delete_unit.

The structure is complete at this point. Note the similarity of program functionalites in this design when compared to that described in the second attempt. The similarity for data structure can be noted also.

The overall machine structure is shown in Figure 13 and the documentation of this design is presented in Appendix C.

CRITIQUE

Utility of the Method

The major drawback of a method which involves iteration on a number of designs and redesigns is the difficulty of deciding when to stop. An iterative process must be supported by a definitive set of procedures which guide the designer. Admittedly, intuition may be the best tool available to the designer. In that regard, the designers of this demonstration problem found their intuition very helpful, but often in conflict with the abstract machine approach. On the contrary, those very familiar with abstract machines relied heavily on their intuition without the same conflicts. This leads to the following conclusion: Training is critical, and the necessary amount and duration has been grossly underestimated.

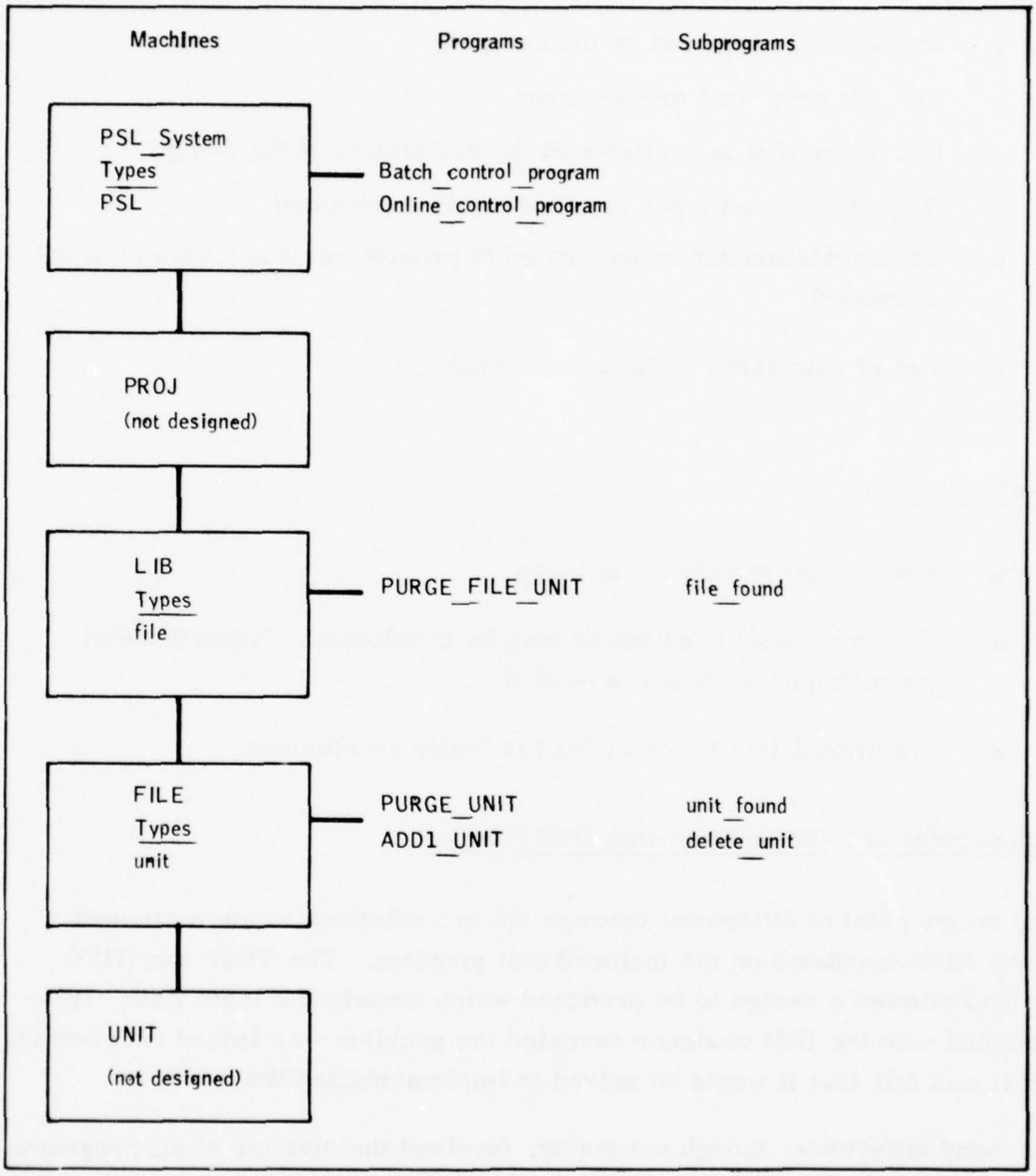


Figure 13. PSL Machines Hierarchy

Strengths and Weaknesses of RDM

Strengths --

- Discipline is imposed on the designer
- Correct programs are designed
- Documentation is available at the completion of the design
- Data objects and types are thoroughly understood
- Documentation can be excerpted to provide various levels of detail as needed
- Use of automated tools is facilitated.

Weaknesses --

- The method is difficult to learn
- Implementation constraints may be overlooked. Experience of actual implementation is needed.
- Traditional intuition may lead to faulty conclusions.

Comparison of RDM Solution with IBM Solution

The major point of difference between the two solutions is the increased detail RDM mandated on the included unit problem. The TDSP and HIPO method allowed a design to be produced which contained a logic flaw. (Discussions with the IBM designer revealed the problem was indeed understood, but it was felt that it would be solved at implementation time.)

A second difference, though not major, involved the division of all programs into batch oriented and on-line oriented at a very high level. This division appears to be unnecessary in the context of RDM, though it is not a design flaw.

6. SUMMARY AND CONCLUSION

In this report the general problems of software development and the foundations for their solutions have been discussed. It has been observed that the difficulties of software development are primarily due to a general lack of design principles leading to a rational design methodology. A methodology is a set of procedures supported by principles which are made effective by a set of notational and organizational tools.

An extensive investigation has been made in the "Design Principles" section at the beginning of Chapter 2 of the research involving three design principles denoted as proof-of-correctness, limiting complexity, and constructive design. It was shown that a set of concepts and functions forming the foundations of a rational design methodology are derived from this research. They include execution-independent (static) representation of the design, notations for design specifications which permit proof-of-correctness, abstraction as a basis for program decomposition, hierarchical interfaces, and constructive design disciplines which allow the designer to arrive at a design specification from a set of requirements by using correctness-preserving design steps.

In the section on "RDM Procedures," chapter 3, the conventions and procedures in existing design methodologies were discussed. It was observed that the top-down, structured programming approach first discussed by Mills and Baker has been the most complete and most extensively described methodology to date. Although this methodology has achieved some popularity among software development organizations, it often has led to less than totally successful results. This is due in part to deficiencies in the methodology. Current research on methodologies generally offer variations on the top down, structured programming approach. Chapter 3 then presented the suggested procedures and conventions of a rational design methodology.

The tools and techniques of a software development methodology were discussed in chapter 4. It was observed that they include notation, clerical

aids, verification techniques, and management schemes to support the design process. The section on notations included discussions of charts, implementation languages, design specification languages, and requirements specification languages. Clerical aids involve computer support tools which relieve the designer of error prone tasks and provide a basis for control and management of the design process. Verification of the design involves both syntactic and semantic checking of the design as well as its performance assessment. Design management includes organization techniques, procedures, and computer support systems to analyze and control the design process. Design Analysis tools provide semi-automatic aids to the designer for verifying the correctness and consistency of the design. Recommendations are made to the RDM user for tools implementation.

Chapter 5 presents the demonstration of the application of rational design methodology to the design of a portion of the IBM developed Program Support Library. The complete documentation of the demonstration design is presented in Appendix C and a complete BNF description of the PDL is presented in Appendix B.

Although the design process is only a phase of software development, it is in this phase that the proper basis for controlled development is established. No implementation tool, programming language, or management technique is effective if it does not interface with or make use of results of the design phase.

The RDM presented here represents an approach to the solution of the design problem. The methodology, however, still requires additional development. Areas where this development appears most profitable are education, performance evaluation, and extensions to other phases of software development including requirements, specifications, implementation, and management. Also, the problem of concurrent processes must be considered. Software development problems can be solved by uncoordinated activities in the various phases of the development process. However, to achieve maximum effectiveness, these efforts should be conceived and planned within the context of the RDM.

7. REFERENCES

[ADAM75] J. M. Adams, "A General, Verifiable Iterative Control Structure," IEEE Transactions on Software Engineering, vol. SE-1, no. 3 (March 1975).

[ALFO76] M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," Procedings, 2nd International Conference on Software Engineering (October 1976).

[BAKE72] F. T. Baker, "Chief Programmer Team: Management of Production Programming," IBM Systems Journal, vol. 11, no. 1 (Jaunary 1972).

[BASU75] S. K. Basu, and J. Misra, "Proving Loop Programs," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March 1975).

[BELL77] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, vol. SE-3, no. 1 (January 1977).

[BOEH75] B. W. Boehm, R. K. McClean, and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March 1975), pp. 125-138.

[BOEH77] B. W. Boehm, "Software Engineering: R&D Trends and Defense Needs," Research Directions in Software Technology, (October, 1977).

[BOHM66] C. Bohm, and G. Jacopini, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," CACM, vol. 9, no. 5 (May 1966), pp. 366-371.

[BOYD76] D. L. Boyd, and G. J. Gustafson, The Design Methodology WELLMADE and its Relationship to the Software Generation Process: An Overview, *Honeywell CRC Report (October 1976).*

[BOYD77] D. L. Boyd, "WELLMADE Design Methodology," Proceedings, Honeywell Software Productivity Symposium, Minneapolis (April 1977).

[BOYD78] D. L. Boyd, and A. Pizzarello: "Introduction to THE WELLMADE design methodology." 3rd Proceedings, International Conference on Software Engineering (May, 1975).

[BRAD72] F. T. Bradshaw, "Some Structural Ideas for Computer Systems," Compcon 1972.

[BRIN73] P. Brinch-Hanson, Operating Systems Principles, Prentice-Hall, Englewood Cliffs , NJ (July 1973).

[BRIN76] ---, "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, vol. SE-2, no. 4 (December 1976).

[BURS69] R. M. Burstall, "Proving Properties of Programs by Structural Induction," Computing Journal, vol. 12 (February 1969), pp. 41-48.

[CHU76] Y. Chu, "Introducing a Software Design Language," Proceedings, 2nd International Conference on Software Engineering, San Francisco (October 1976), pp. 291-304.

[CLIN70] M. Clint, and C. A. R. Hoare, "Program Proving: Jumps and Functions," Acta Informatica, 1 (1970), pp. 214-224.

[CLIN73] M. Clint, "Program Proving: Coroutines," Acta Informatica, 2 (1973), pp. 50-63.

[DAHL68] O.-J. Dahl, B. Myhrhaug,and K. Nygaard, The Simula 67 Common Base Language, Publication S-22, Norwegian Computing Centre, Oslo, Norway (1968).

[DAHL72] O.-J. Dahl, C. A. R. Hoare, "Hierarchical Program Structures," Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, New York, NY (1972).

[DAVI77] C. G. Davis, and C. R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, vol, SE-3, no. 1 (January 1977), pp. 69-84.

[DEC69] DEC, PDP-10 Reference Handbook, Code AIW, Digital Equipment Corp. (1969).

[DEC76] ---, DECSystem-200 Users Guide, DEC-20-OUGAA-A-D, Digital Equipment Corp. (October 1976).

[DERS76] N. Dershowitz, and Z. Manna, The Evolution of Programs: A System for Automatic Program Modification, Computer Science Department Report, Stanford University (December 1976).

[DERS77] ---, Automatic Program Annotation, ARPA Contract MDA 903-76-C-0206 and AFOSR Grant AFOSR-76-2902 (March 1977).

[DIJK68] E. W. Dijkstra, "The Structure of the "THE" - Multiprogramming System," CACM, vol. 11, no. 5 (May 1968), pp. 341-346.

[DIJK72] ---, "Notes on Structured Programming," Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, New York, NY (1972).

[DIJK75] ---, "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs," CACM, vol. 18, no. 8 (August 1975).

[DIJK76] ---, A Discipline of Programming, Prentice-Hall Inc., Englewood Cliffs, NJ (1976).

[ELSP72] B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, vol. 4 (June 1972).

[ERNS77] G. W. Ernst, "Rules of Inference for Procedure Calls," Acta Informatica, 8 (1977), pp. 145-152.

[FISH74] D. A. Fisher, "Automatic Data Processing Costs in the Defense Department," Institute for Defense Analysis, P. 1046, (1972).

[FLOY67] R. W. Floyd, "Assigning Meanings to Programs," Proceedings, Symposium in Applied Mathematics, vol. 19, J. T. Schwartz (ed.), American Mathematical Society, Providence, RI (1967), pp. 19-32.

[GERM75] S. M. German, and B. Wegbreit, "A Synthesizer of Inductive Assertions," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March 1975), pp. 68-75.

[GERH76] S. L. Gerhart, L. Yelowitz, "Observation of Fallibility in Applications of Modern Programming Methodologies," IEEE Trans. on Software Engineering, SE-2, 3 Sept. 1976.

[GLAS72] E. L. Glaser, "Introduction and Overview of the LOGOS Project," Compcon 1972.

[GOLD63] H. H. Goldstine, and J. von Neumann, "Planning and Coding Problems for an Electronic Computer Instrument," Collected Works of John von Neumann, vol. 5, A. M. Traub (ed.), Pergammon Press, New York (1963), pp. 80-235.

[GOOD75] D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Program Verification System," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March 1975).

[GRAH73] R. M. Graham, G. J. Clancy and D. B. Devaney, "A Software Design and Evaluation System," CACM, vol. 16, no. 2 (February 1973).

[GRIE76] D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," IEEE Transactions on Software Engineering, vol. SE-2, no. 4 (December 1976).

[GRIE77] D. Gries, and N. Gehani, "Some Ideas on Data Types in High-Level Languages," CACM, vol. 20, no. 6 (June 1977).

[GUTT77] J. Guttag, "Abstract Data Types and the Development of Data Structures," CACM, vol. 20, no. 6 (June 1977).

[HAMI76] M. Hamilton, and S. Zeldin, "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, vol. SE-2, no. 1 (March 1976), pp. 9-32.

[HOAR69] C. A. R. Hoare, "An Axiomatic Approach to Computer Programming," CACM, vol. 12, no. 10 (October 1969(, pp. 576-580, 583.

[HOAR71] ---, "Procedures and Parameters: An Axiomatic Approach," Lecture Notes in Mathematics 188, (E. Engeler (ed.), Symposium on the Semantics of Algorithmic Languages), Springer, Berlin-Heidelberg-New York (1971), pp. 102-106.

[HOAR72a] ---, "Proof of Correctness of Data Representation," Acta Informatica, 1 (1972), pp. 271-281.

[HOAR72b] ---, "Notes on Data Structuring," Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, New York, NY (1972).

[HOAR72c] ---, "Towards a Theory of Parallel Programming," Operating System Techniques, C. A. R. Hoare and R. H. Perrot (eds.), Academic Press, New York, NY (1972).

[HOAR73] ---, and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, 2 (1973), pp. 335-55.

[HOAR74] ---, "Monitors: An Operating System Structuring Concept," CACM, vol. 17, no. 10 (October 1974).

[HONE77a] Honeywell, Rational Design Methodology, Status Report no. 2, RADC Contract no. F30602-77-C-0043.

[HONE77b] ---, Software Engineering Handbook, Internal Publication, Honeywell Large Information System Division, Phoenix, AZ (June 17, 1977).

[IBM73] IBM, HIPO: Design Aid and Documentation Tool, IBM SR20-9413-0, IBM Corporation, Poughkeepsie, NY (April 1973).

[ICHB74] J. D. Ichbiah, et. al., The System Implementation Language LIS, CII, 68 Route De Versailles, 78430 Louveciennes, France (December 1974).

[IRVI77] C. A. Irvine, and J. W. Brackett, "Automated Software Engineering Through Structured Data Management," IEEE Transactions on Software Engineering, vol. SE-3 no. 1 (January 1977).

[JACK75] M. Jackson, Principles of Program Design, Academic Press (1975).

[JENS75] K. Jensen, and N. Wirth, Pascal User Manual and Report, 2nd Edition, Springer-Verlag (1975).

[KARA77] J. Karas, D. Mackellar, R. Hickey, D. Kayden, and T. Carlin, "Structured Programming Techniques Applied to Operating Systems Development: Honeywell Series 6000 Clear Memory Utility," Honeywell Software Productivity Symposium, (April 26-28, 1977), pp. 6.2.1-6.2.20.

[KATZ75] S. Katz, and Z. Manna, "Towards Automatic Debugging of Programs," SIGPLAN Notices, vol. 10, no. 6 (Proceedings, 1975 International Conference on Reliable Software) (June 1975), pp. 143-155.

[KATZ77] S. Katz, and Z. Manna, "Logical Analysis of Programs," CACM, vol. 19, no. 4 (April 1977), pp. 188-206.

[KING71] J. C. King, "A Program Verifier," Proceedings, IFIP Congress 1971 (1971), pp. 234-249.

[KNUT74] D. E. Knuth, "Structured Programming With GOTO Statements," Current Trends in Programming Methodology, vol, 1, Yeh (ed.), Prentice-Hall, Englewood Cliffs, NJ (1977), pp. 140-194; also ACM Computing Surveys, vol, 6, no. 4 (December 1974).

[KNUTH] D. E. Knuth, "The Art of Computer Programming," Vol. 1, 2, 3.

[KOWA77] T. Kowaltowski, "Axiomatic Approach to Side Effects and General Jumps," Acta Informatica, 7 (1977), pp. 357-360.

[LAMP77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the Programming Language Euclid," SIGPLAN Notices, vol. 12, no. 2 (February 1977).

[LISK72] B. H. Liskov, "A Design Methodology for Reliable Software Systems," FJCC 72 (1972), pp. 191-99.

[LISK73] B. H. Liskov, Guidelines for Design and Implementation of Reliable Software Systems, MITRE Corp., Bedfore, Mass. (February 1973).

[LISK74] B. H. Liskov, A Note on CLU, Computation Structures Group Memo 112, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass. (November 1974).

[LISK75] B. H. Liskov, and S. Zilles, "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March 1975).

[LOGI76] Logicon, "An Overview of Software Development and Management," Management Guide to Avionics Software Acquisition, vol. 1 (June 1976), ASD-TR-76-11.

[LOND70] R. L. London, "Proving Programs Correct: Some Techniques and Examples," BIT, 10 (1970), pp. 168-182.

[LUCK77] D. C. Luckham, and N. Suzuki, "Proof of Termination Within a Weak Logic of Programs," Acta Informatica, 8 (1977), pp. 21-36.

[MANN69] Z. Manna, "The Correctness of Programs," J. CSS, vol. 3, no. 2 (May 1969), pp. 119-127.

[MANN74a] ---, Mathematical Theory of Computation, McGraw-Hill (1974).

[MANN74b] ---, and A. Pnueli, "Axiomatic Approach to Total Correctness of Programs," Acta Informatica, 3 (1974), pp. 243-263.

[MANN76] ---, and R. Waldinger, "Is 'Sometimes' Sometimes Better Than 'Always'? Intermittent Assertions in Proving Program Correctness," Proceedings, 2nd Conference on Software Engineering, San Francisco (October 1976).

[MCCA63] J. McCarthy, "Towards a Mathematical Science of Computation," Proceedings, IFIP Congress 1962, Munich, Germany, North-Holland Publ. Co., Amsterdam (1963), pp. 21-28.

[MCGE76], Use of Rational Programming Techniques for the Development of Reliable Software, Honeywell HIS/ECO Internal Report (1976).

[MCGE77] R.C. McGee, and A. Pizzarello, "Software Development Methodology at LSEO-Phoenix," Proceedings, Honeywell Software Productivity Symposium, Minneapolis, MN (April 1977).

[MILL76] J.K. Millen, "Security Kernel Validation in Practice," CACM, Vol. 19, No. 5 (May 1976).

[MILL71] H.D. Mills, "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, NJ (1971), pp. 41-55.

[MILL72a] ---, Mathematical Foundations for Structured Programming, Report no. FSC 72-6012, IBM Federal Systems Division, Gaithersburg, MD (February 1972).

[MILL72b] ---, Chief Programmer Teams: Principles and Procedures, Report no. FSC 71-5108, IBM Federal Systems Division, Gaithersburg, MD (June 1972).

[MILL73a] ---, How to Write Correct Programs and Know It, Report no. FSC 73-5008, IBM Federal Systems Division, Gaithersburg, MD (February 1973).

[MILL73b] ---, "On the Development of Large Reliable Programs," Proceedings, 1973 IEEE Symposium on Computer Software Reliability, New York, NY (April 30-May 2, 1973).

[MILL74] ---, "Techniques for the Specification and Design of Complex Programs," Proceedings, Third Texas Conference on Computing Systems, Austin, TX (November 1974).

[MILL75] ---, "The New Math of Computer Programming," CACM, vol. 18, no. 1 (January 1975).

[MORR77] J. H. Morris, and B. Wegbreit, "Subgoal Induction," CACM, vol. 20, no. 4 (April 1977).

[MYER75] G. J. Myers, Reliable Software Through Composite Design, New York, Petrocelli/Charter (1975).

[NASS73] I. Nassi, and B. Schneiderman, "Flowchart Techniques for Structured Programming," SIGPLAN Notices, vol. 8, no. 8 (August 1973).

[NATO69] NATO, NATO Science Committee Report on Software Engineering, Randell and Naur (eds.), NATO Science Affairs Division, Brussels, 39, Belgium (January 1969).

[NAUR69] P. Naur, "Programming by Action Clusters," BIT, 9 (1969), pp. 250-58.

[NOON75] R. F. Noonan, "Structured Programming and Formal Specifications," IEEE Transactions on Software Engineering, vol. SE-1, no. 4 (December 1975).

[ORGA72] E. I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, MA (1972).

[OWIC76] S. Owicki, and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica, 6 (1976).

[PARN72a] D. L. Parnas, "A Technique for the Specification of Software Modules With Examples," CACM, vol. 15, no. 5 (May 1972), pp. 330-336.

[PARN72b] ---, "On the Criteria to be Used in Decomposing Systems Into Modules," CACM, vol. 15, no. 12 (December 1972), pp. 1053-1058.

[PIZZ76] A. Pizzarello, Concurrent Processing, Honeywell LSEO Report (1976).

[PIZZ77] ---, The Constructive Approach, Honeywell LSEO Internal Report (1977).

[RABI77] M. O. Rabin, Complexity of Computation, CACM 20, 9.

[RADC] RADC, Structured Programming Series, RADC-TR-74-300. *

[REIF75] D. J. Reifer, "Automated Aids for Reliable Software," Proceedings, International Conference on Reliable Software (April 1975).

[REYN76] C. Reynolds, and R. T. Yeh, "Induction as the Basis for Program Verification," IEEE Transactions on Software Engineering, vol. SE-2, no. 4 (December 1976), pp. 244-252.

[ROBI77] L. Robinson, and K. N. Levitt, "Proof Techniques for Hierarchially Structured Programs," CACM, vol. 20, no. 4 (April 1977), pp. 271-283.

[ROSE72] C. W. Rose, F. T. Bradshaw, and S. W. Katzke, "The LOGOS Representational System," Compcon 1972.

[ROSS77] D. T. Ross, and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, vol. SE-3, no. 1 (January 1977), pp. 6-15.

[ROUB77] O. Roubine, and L. Robinson, Special Reference Manual, 3rd Edition, Stanford Research Institute Technical Report CSG-45, Stanford Research Institute (1977).

[SIGP72] SIGPLAN, "The GOTO Controversy," SIGPLAN Notices, vol. 7, no. 11 (Special Issue on Control Structures in Programming Languages) (November 1972).

[STEV74] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, vol. 13, no. 2 (1974).

[TAMI76] M. Tamir, ADI - Automatic Derivation of Invariants, Masters Thesis, Weizmann Institute of Science, Rehovot, Israel (August 1976).

[TAUS77] R. C. Tausworthe, Standardized Development of Computer Software, Prentice-Hall Inc., Englewood Cliffs, NJ (1977).

[TEIC77] D. Teichrow, and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, vol. SE-3, no. 1 (January 1977).

[TRAI73] W. L. Trainor, "Software: From Satan to Savior," Proceedings, NAECON (May 1973).

[TURN76] R. Turn, M. R. Davis, and R. N. Reinstedt, "A Management Approach to the Development of Computer-Based Systems," Proceedings, 2nd International Conference on Software Engineering, San Francisco, CA (13-15 October 1976), pp. 305-311.

[VANL76] P. Van Leer, "Top-Down Development Using a Program Design Language," IBM Systems Journal, vol. 15, no. 2 (1976), pp. 155-170.

[WALD73] R. J. Waldinger, and K. N. Levitt, "Reasoning About Programs," Proceedings, Symposium on Principles of Programming Languages, Boston, MA (October 1973).

[WARN] J. D. Warnier, and B. M. Flanagan, Entrainement a la Programmation, Tome 1: Construction de Programme, Tome 2: Exploitation des donnee, Les Editions d'organization, Paris.

[WEGB74] B. Wegbreit, "The Synthesis of Loop Predicates," CACM, vol. 17, no. 2 (February 1974), pp. 102-112.

[WEGB76] B. Wegbreit, and J. M. Spitzen, "Proving Properties of Complex Data Structures, JACM, vol. 23, no. 2 (April 1976), pp. 389-396.

[WEGB77] B. Wegbreit, "Constructive Methods in Program Verification," IEEE Transactions on Software Engineering, vol. SE-3, no. 3 (May 1977), pp. 193-209.

[WEGB77a] B. Wegbreit, "Verifying Program Performances," JACM 23, 4.

[WIRT71] N. Wirth, "Program Development by Stepwise Refinement," CACM, vol. 14, no. 4 (April 1971), pp. 221-227.

[WIRT77] N. Wirth, "Modula: A Language for Modular Multiprogramming," Software - Practice and Experience, vol. 7, no. 1 (January 1977).

[WIRTH] N. Wirth, "Algorithms + Data Structure = Programs," Prentice-Hall.

[WULF76] W. A. Wulf, R. L. London, and M. Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs," IEEE Transactions on Software Engineering, vol. SE-2, no. 4 (December 1976).

[YEH77] R. T. Yeh, "Verifying Programs by Predicate Transformations," Infotech State of the Art Report on Software Engineering Techniques (1977).

[ZAHN74] C. T. Zahn, "A Control Statement for Natural Top-Down Structured Programming," Lecture Notes in Computer Science, No. 19 (Symposium on Programming Languages, Paris, France, 1974), Springer-Verlag, New York.

[ZILL76] S. N. Zilles, Data Algebra: A Specification Technique for Data Structures, Ph. D. Thesis (forthcoming), Project MAC, Massachusetts.

* 74-300, Vol I A016771, 74-300, Vol II A018046, 74-300, Vol III A013255, 74-300, Vol IV A015794, 74-300, Vol V A003339, 74-300, Vol VI A007796, 74-300, Vol VII A008639, 74-300, Vol VII Addendum A016414, 74-300, Vol VIII A016415, 74-300, Vol IX A008640, 74-300 Vol X A008861, 74-300, Vol XI A016416, 74-300, Vol XII A026947, 74-300, Vol XIII A020858, 74-300, Vol XIV A015795, 74-300, Vol XV A016668.

Appendix A - PDL Description

The syntax and informal descriptions of the semantics of the program design language (PDL) will be presented in this appendix. The PDL includes notation for general design specifications, detailed design specifications, and documentation of the total design. The notations include program flow constructs, expressions, primitive data structures, constructs for data structure descriptions, and object definitions. The syntatic description will be presented as BNF productions. The discussion on semantics will include informal descriptions of all constructs.

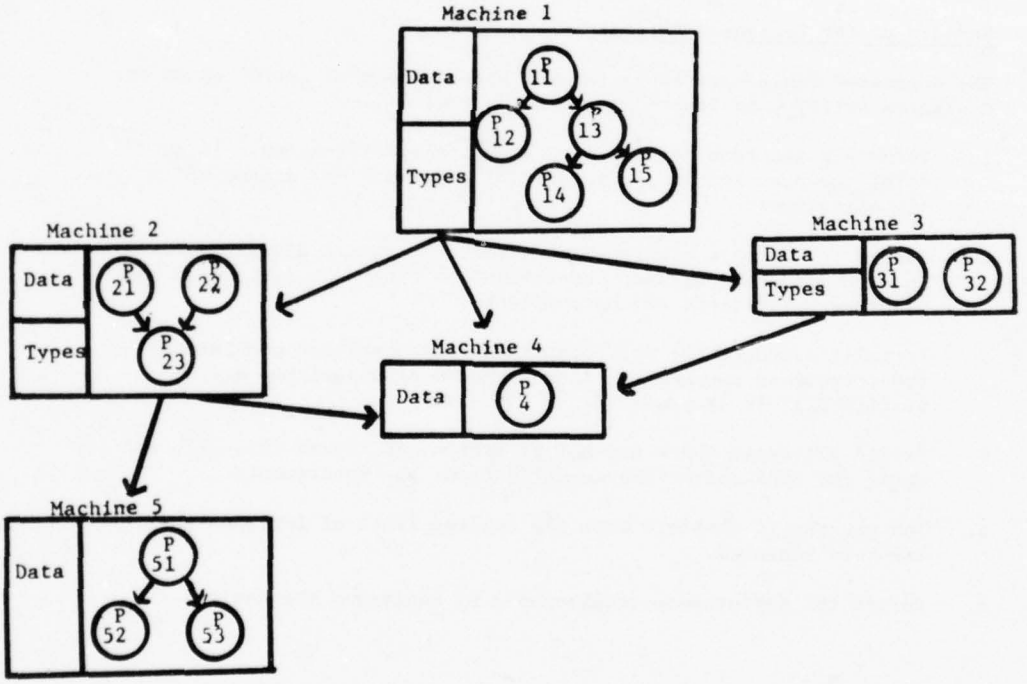


Figure A.1: Hierarchy of Abstract Machines

NOTATION CONVENTIONS

O, A, B, X, Y are object symbols
t is a type symbol
F is a function symbol
V denotes a value
. and : are used to qualify symbols
{see "Dot and Colon Notation"}
NOTE: In addition, the colon (:) is used to separate a symbol and its definition
{see "Type Declarations" and "Data (Variable) Declarations"}
[ ] information between brackets is optional
< > denotes a syntactic symbol
{ } denotes explanatory information
<text> denotes English text

Summary of the Designers Activity

The suggested design procedure for software systems as reflected in the designers activity is described briefly below.

1. Translate the requirements into a static specification. In so doing the data types must be partially defined (as a name and a set of values).

2. Design one or more programs to reflect the static design from (1). In so doing, the necessary operations on types are discovered and the type definitions may be completed.

3. Formally document the work done thus far, check for consistency, and correct as necessary. Formal proofs of algorithms may be performed if it is required.

4. Expand all types which are not primitive and repeat (1), (2), and (3) using the type operations specifications as requirements.

5. The process is complete when the desired level of implementability has been achieved.

6. Verify the performance requirements by analyzing the design.

DATA STRUCTURES {V-NOTATION}

1. PRIMITIVE DATA TYPES
   and their operations

   a. integer
      {unary} -, {unary} +,
      +, -, *, ÷, =, ≠, <, ≤, >, ≥, :=

   b. real
      {unary} -, {unary} +,
      +, -, *, /, =, ≠, <, ≤, >, ≥, :=

   c. boolean
      {constant}T, {constant}F
      not, and, or, xor, cand, cor, :=, =, ≠

   d. char
      =, ≠, <, ≤, >, ≥, :=

   e. (lv..hv) {subrange}
      where lv and hv are lowest value and
      highest value of a primitive type

   f. (V1▮V2▮...▮Vn) {enumerated}
      where V1, V2, ..., Vn are the only
      permissible values of this data type
      {eg., COLOR: (RED▮WHITE▮BLUE)}

   g. abstract {a type designator whose
      definition is given elsewhere}

2. PRIMITIVE DATA STRUCTURES
   and their operations

   a. Cartesian Product {Record}

      ordered $(O_1:t_1;\ O_2:t_2;\ \ldots:\ O_n:t_n)$

      unordered $(O_1:t_1,\ O_2:t_2,\ \ldots,\ O_n:t_n)$

      selector operation:
      if A is an ordered or unordered
      cartesian product then $A.O_i$ or
      $A:O_i$ selects the ith component

   b. Discriminate Union

      $(V_1:t_1▮V_2:t_2▮\ldots▮V_n;t_n)$

      tag operation:
      if A is a discriminate union
      then the variable A.tag has the
      value V_i if the value of A is
      of type t_i

c. Array {or sequence}

t array

Operations

if A is an array of items of type t then

A.lob - the integer which is the smallest index

A.hib - the integer which is the largest index

A.dom - the non-negative integer which is the number of elements; A.dom=A.hib-A.lob+1≥0

A(I) - the value of the Ith item

A.low - the first item

A.high - the last item

A:lorem - remove the first item

A:hirem - remove the last item

A:loext(a) - append a new first item, a

A:hiext(a) - append a new last item, a

$A:=(k,a_1,a_2,\ldots,a_n)$ - initialize A with A.lob=k, and $A(k)=a_1$, $A(k+1)=a_2,\ldots, A(k+n-1)=a_n$

A:swap(I,J) - exchange the Ith and Jth item

A:shift(n) - shift the domain n places, n an integer. (n>0 to the right, n<0 to the left)

3. Dot and Colon Notation

. - Object.function (parameters)
value producing operation on a data structure

Eg. A.dom, A.lob, B.tag

: - Object:function (parameters)
value changing operation on a data structure

Eg. A:loext, A:swap(3,4)

4. Data (Variable) Declarations

a. Format

<object id>:<type expression>[<scope>]
{<text>}

Eg. X:integer

b. Scope - applies only to variable of programs within a single module
p - (private), local and uninitialized
l - (latent), inherited and uninitialized
g - (global), inherited and initialized
c - (constant), v-(variable)

Scoping rules pc, pv, lc, lv, gc, gv

Eg. X:integer pv
{X is private variable integer}

5. Type Declarations

```
<type name>[(<parameter list>)]:<type expression>;
      [let{<data declarations>}:]
      [inv{<invariant assertion>}:]
      [fun<function name>(<function parameters>)
            [returns<object>:<type>]
            [req<specifications>];]
      ⋮
[end<type name>]
```

where <parameter list>, <data declarations>, and <function parameters> is an unordered cartesian product; <type expression> is an expression of primitive structures and types; and <invariant assertion> and <specifications> are assertions

Eg.

```
stack(n:integer):name array
    let {S:stack(n), n: integer}
    inv {S.dom ≤ n}
    fun push(X:name) returns S'
          req {in:S.dom < n,
               out: S'=S:hiext(X)}
    ⋮
    fun full returns b:boolean
          req {(S.dom=n ⇒ b=true) or
               (S.dom<n ⇒ b=false)}
end stack
```

PROGRAM STRUCTURES {P-NOTATION}

1. Sequential operations

   S_1; S_2 {S_i-statements}

2. No operation

   skip

3. Assignment

   $X_1, X_2, \ldots, X_n := E_1, E_2, \ldots, E_n$

   The value of the expression, E_i, is assigned to be the value of the variable, X_i. Assignment is made in parallel to all X_i, i=1,2,...n

4. Selection

   if $B_1 \rightarrow S_1 \blacksquare B_2 \rightarrow S_2 \blacksquare \ldots \blacksquare B_n \rightarrow S_n$ fi

   where $B_1, B_2, \ldots, B_n$ are boolean-valued expressions called the guards
   and $S_1, S_2, \ldots, S_n$ are program statement lists called the guarded commands
   A true guard, B_i, is selected and the corresponding guarded command S_i is executed. At least one guard must be true, else the program aborts.

5. Iteration

   do $B_1 \rightarrow S_1 \blacksquare B_2 \rightarrow S_2 \blacksquare \ldots \blacksquare B_n \rightarrow S_n$ od

   where $B_1, B_2, \ldots, B_n$ are boolean-valued expressions called the guards
   and $S_1, S_2, \ldots, S_n$ are program statement lists called the guarded commands
   A true guard, B_i, is selected, the corresponding guarded command is executed and then the process repeated. When all guards are false, a skip occurs.

6. Functions

   Data structure operations

infix	(x + y)
dot	A.low
colon	A.hiext(I)

   Programs

   <program name>(<actual parameters>)

   Initialization

   Object:init(<actual parameters>)

7. Comments

   {<text>}

8. Assertions

   Assert{<assertions>}

SPECIFICATIONS

Notation for assertions, invariants, requirements, conditions, and input, output, or state specifications.

Forms

1. English text {<text>}
2. Effects of (value of) data structure operations.
3. Predicate Definitions:

   <predicate>(<parameters>)means<text>

4. Mathematical Symbols:

   a. Logical connectors

      or (|,V), and (∧,&), not (¬), exclusive or (xor), conditional and (cand), conditional or (cor), then (implies,⇒)

   b. Quantifiers

      existential - (∃,E, there exist)
      universal - (∀,A, for all)

   c. Sets

      element (ε), Union (∪), intersection (∩)
      difference (A\B), Complement ($\bar{A}$)

   d. Primed symbols denote values after the execution of the program; whereas non-primed symbols denote values before execution.

```
DOCUMENTATION

REQUIREMENTS INTERPRETATIONS <name>:
    <text>
SYSTEM <name>
    <machine name>{<text>}
    <machine name>{<text>}
      :

MACHINE <machine name>:
  MISSION DESCRIPTION
      Functional Specification
          {<text>}
      Usage Information
          {<text>}
      Acceptance Criteria
          {<text>}
  DESIGN DOCUMENTATION
      Defines <type name list>
      Overview
          {<text>}
      Predicates
          <predicate definitions>;
            :
          <predicate definitions>
      Types
          <type declarations>:
            :
          <type declarations>
      Permanent Data
          <data declaration>;
            :
          <data declaration>
      Data Invariant
          {<assertions>}
      Initialization
          <program text>
      Programs
          <program name>{<text>}
            :
          <program name>{<text>}
```

```
(SUB) PROGRAM <program name>[(<param. list>)] [returns <param.>]
        Overview
            {<text>}
        Variables
            <variable declaration><scope>
             ⋮
            <variable declaration><scope>
        Subprograms
            <program name>{<text>}
             ⋮
            <program name>{<text>}
        Input Specifications   {assertion}
        Output Specifications   {assertion}
        Performance Specifications  {assertion}
        Invariant   {assertions}
        Text
            <program text>
        Notes
            {<text>}
        END <program name>
(SUB)PROGRAM program name [( param. list>)] {[returns <param.>]}
    ⋮
END<machine name>


MACHINE  <machine name>
   ⋮
END<machine name>
   ⋮
END <system name>
```

Appendix B - RDM Notation BNF

DOCUMENTATION LEVEL SYNTAX

REQUIREMENTS

```
<system_design>     ::=
     <requirements>
     <system_desc>

<requirements>      ::=
      REQUIREMENTS INTERPRETATIONS  <system_id> :
         <text>
```

SYSTEM

```
<system_desc>       ::=
     SYSTEM  <system_id>  :
             <machine_index>
             <machine_doc_list>

<machine_index>  ::=
     <machine_id> <text> <machine_index> |
     <machine_id> <text>

<machine_doc_list>  ::=
    <machine_doc> <machine_doc_list> |
    <machine_doc>
```

MACHINE

```
<machine_doc>  ::=
     MACHINE  <machine name>:
     MISSION DESCRIPTION
          <mission_desc>
     DESIGN DOCUMENTATION
          <design_doc>

<mission_desc>  ::=
     FUNCTIONAL DESCRIPTION
          <text>
     USAGE INFORMATION
          <text>
     ACCEPTANCE CRITERIA
          <text>

<design_doc>  ::=
     <machine_design_spec>
     END  <machine_id>
```

MACHINE

```
<machine_design_spec> ::=
          DEFINES <type_id_list> ;
     <machine_static_design>
     <machine_program_design>
```

MACHINE STATIC DESIGN

```
<machine_static_design> ::=
     <overview>
     <predicates>
     <types>
     <data>
     <invariant>
     <initialization>
     <programs> ;

<types >::=
     ϕ |
     types <type_dec_list> ;

<predicates> ::=
     ϕ |
     Predicates <text> ;

<data> ::=
     ϕ |
     Permanent data <object_dec_list> ;

<invariant> ::=
     ϕ |
     Data Invariant <text>;

<initialization>::=
     ϕ |
     Initialization <program text>;

<programs> ::=
     ϕ |
     Programs <program_dec_list> ;
```

TYPES DECLARATION

```
<type_dec_list> ::=
     <type_dec>  <type_dec_list>  |
     <type_dec>

<type_dec> ::=
     <type_id> <type_parameters>:
               <type_expression>
               <local_objects>
               <invariants >
               <functions>
      end <type_id>;

<type_parameters> ::=
      ϕ |
      (<formal_parameters>)

<type_expression> ::=
     <type_id> |
     <type_id> (<actual_parameters>) |
      abstract |
     <prim_data_type>  |
     <structured_type>

<prim_data_type> ::=
      boolean |
      integer |
      real    |
      char    |
      (<subrange>) |
      (<enumerated>)

<subrange>    ::=
     <lb>..<ub>

<lb>  ::=
     <integer_constant>

<ub>  ::=
     <integer_constant>

<enumerated >   ::=
     <primitive_value>▮<enumerated>|
     <primitive_value>

<structured_type>  ::=
     <array> |
     <ordered_cartesian_product> |
     <unordered_cartesian_product> |
     <union>
```

```
<array> ::=
     <type_expression> array

<union> ::=
     <structure_head>•<union> |
     <structure_head>

<structure_head> ::=
     <id>:<type_expression>

<ordered_cartesian_product> ::=
     <structure_head>;<ordered_cartesian_product> |
     <structure_head>

<unordered_cartesian_product> ::=
     <structure_head>,<unordered_cartesian_product> |
     <structure_head>

<local_objects >::=
      ϕ |
      let <object_dec_list>;

<function> ::=
      function <function_id><input_parameters><output_parameters>
               req  <text>;

<input_parameters> ::=
      ϕ |
      (<formal_parameters>);

<output_parameters> ::=
      ϕ |
      returns <formal_parameters>
```

DATA DECLARATIONS

```
<object_dec_list> ::=
     <object_dec><object_dec_list> |
     <object_dec>

<object_dec> ::=
     <object_id_list>:<type_expression>;

<object_id_list>::=
     <object_id>,<object_id_list>|
     <object_id>
```

PROGRAMS

```
<program_dec_list>::=
     <program_dec>;<program_dec_list> |
     <program_dec>

<program_dec >::=
     <program_id><text>
```

PARAMETERS

```
<formal_parameters> ::=
     <object_dec_list>

<actual_parameters> ::=
     <value_ref>,<actual_parameters> |
     <value_ref>
```

PROGRAM (DETAILED) DESIGN SYNTAX

PROGRAM

```
<machine_program_design > ::=
     <program_list>

<program_list> ::=
     <program> <program_list> |
     <program>

<program> ::=
      PROGRAM  <program_doc> |
      SUBPROGRAM  <program_doc>

<program_doc> ::=
     <program_id>:<input_parameters><output_parameters>;
     <overview>
     <variables>
     <subprograms>
     <specifications>
     <loop_invariant>
     <program_text>
     <notes>
end  <program_id>;

<overview> ::=
      overview; |
      overview <text> ;

<variables> ::=
      variables; |
      variable <variable_list>;

<subprograms> ::=
      subprograms ; |
      subprograms  <program_dec_list>;

<specifications >::=
      Input Specifications  <text>;
      Output Specifications <text>;
      Performance Specifications <text>;

<loop invariants>::=
      ϕ |
      Loop Invariants<text>;
```

```
<program_text> ::=
      text <p_notation>;

<notes> ::=
      notes; |
      notes <text>;


VARIABLES

<variable_list> ::= <object_dec><scope>;<variable_list> |
     <object_dec><scope>

<scope> ::=
      gv | gc | lv | lc | pv | pc


P-NOTATION

<p_notation>::=
     <statement>;<p_notation> |
     <statement>

<statement> ::=
     <simple_statement> |
     <control_statement>

<control_statement> ::=
     <alternative_construct> |
     <repetitive_construct>

<simple_statement> ::=
     <program_activation> |
      skip |
      assert  {<text>} |
      comment {<text>} |
     <assignment_statement>


ALTERNATIVE CONSTRUCT

<alternative_construct> ::=
      if <guarded_command_set> fi
```

REPETITIVE CONSTRUCT

```
<repetitive_construct> ::=
      do <guarded_command_set> od
```

GUARDED COMMAND SET

```
<guarded_command_set>  ::=
     <guarded_command> | <guarded_command_set> |
     <guarded_command>

<guarded_command> ::=
     <guard>→<p_notation>

<guard> ::=
     <expression>
```

PROGRAM ACTIVATION

```
<program_activation> ::=
      init <activation_parameters> |
     <.funs><activation_parameters> |
     <array_funs>

<.funs> ::=
     <id> |
     <id> : |
     <id> . <.funs> |
     <id> : <.funs>

<activation_parameters> ::=
      ϕ |
      (<actual_parameters>)
```

ARRAY FUNCTIONS

```
<array_funs> ::=
     <.funs>.<array_v_fun><activation_parameters> |
     <.funs>:<array_ov_fun><activation_parameters>

<array_v_fun> ::=
```

```
      hib | lob | dom | high | low

<array_ov_fun> ::=
      $ |shift | swap | loext | hiext | hirem | lorem
```

ASSIGNMENT STATEMENT

```
<assignment_statement> ::=
     <assignment_ref>::= <expression>    - ambiguous with "=" relational op
                                           if "=" also used for assignment.

<assignment_ref> ::=
     <program_activation> |

<value_ref> ::=
     <program_activation> |
     <constant>
      (<expression>)

<expression> ::=
     <term><relational_op><term> |
     <term>

<term> ::=
     <term><adding_op><factor> |
     <factor>

<factor> ::=
     <factor><multiplying_op><unary_term> |
     <unary_term>

<unary_term> ::=
     <unary_op><value_ref>

<relational_op >::=
      = | ≠ | < | ≤ | > | ≥

<adding_op> ::=
      + | - | or | cor

<multiplying_op>  ::=
      * | / | + | and | cand

<unary_op> ::=
      - | + | not
```

REQUIREMENT INTERPRETATIONS PSL_DEMONSTRATION

Provide a tool (collection of procedures) for storing and maintaining the data emanating from and necessary for a programming project, to include:

a. Storage and maintenance of programming data
b. Output of programming data and related control data
c. Support of the compilation and testing of programs
d. Support of the generation of program documentation
e. Support and generation of the PSL itself
f. Collection and reporting of management data
g. Control over the integrity and security of data stored in the library
h. Separate support for the clerical activity related to a programming project.

The actual functions to be provided by each of the categories a-h are:

a: add data, purge data, replace source lines, change source data, copy items from point to point, generate sequence numbers for lines of source data, compress source data. {Only the "purge" function will be completely designed}
b: file directory index output, source data output, programmers directory/listings output, character string scan
c: precompilation interface, linkage interface, execution interface
d: print text units, format listings, page number listings, generate magnetic tape in print image form
e: install a PSL, initial a new project, allocate files, terminate a project or file, maintain the list of authorized users

f: initiate/terminate a management data file, collect management data, automatically report management data, print management data, report data history, check automatic exception, support user routines

g: {unspecified by IBM}

h: {unspecified by IBM}

SYSTEM PSL_DEMONSTRATION

PSL_SYSTEM

PROJ {not designed}

LIB

FILE

UNIT {not designed}

MACHINE PSL_SYSTEM

MISSION DESCRIPTION

Functional Specification. Two control programs (Batch_Control_Program and Online_Control_Program) are envisioned for this machine. For the purpose of this demonstration only the Batch_Control_Program has been designed and only the functions updating the PSL are emphasized.

Usage Information. Requests to perform any of the above functions should be entered through either a batch input stream (JCL cards) or an online request.

Acceptance Criteria. The design should be general enough to be compatible (in terms of data storage) with all target computers.

DESIGN DOCUMENT

Overview

This machine defines a PSL system which may be accessed by batch or on-line requests. This is due to the desire to reproduce as faithfully as possible the existing IBM design. There is no design or implementation constraint which requires this distinction.

Predicate

REMOVED (u) implies that a unit, u, and its accounting information have been taken from the file.

Types

```
unit: abstract
file: abstract
library: abstract
project: abstract
info: abstract
      {rest of the requests}
cmd: (purge▮ add▮ create▮...)
     {any valid PSL command}
request: (C:cmd,I:info)
message: integer
     let m: message
     {prints the status message m on the users device}
     fun print returns m
 end message

 PSL: project array
     let X: PSL, R: request, F: file
     fun exists returns  b: boolean
          req  {not designed in this demonstration: b=T, if PSL exists,
               otherwise=F}
     fun install (R.I:info) returns m: message
          req  {installs a PSL named X such that X.exists=T }
     fun merge (L,L1:library) returns m: message
          req  {merges L1 with L to form a new library L'}
```

```
        fun restore (P:project) returns m: message
              req  {restores P from backup storage}
        fun add (u: unit) returns m: message
              req  {adds the unit u to the file F
                    F' = F: hiext(u)}
        fun purge (u: unit) returns m: message
              req  {in: there exists at most one i,
                    F.lob ≤ i ≤ F.hib, u=F(i)
                    out: REMOVED (u)}
    end PSL
```

Permanent Data

x: PSL

Data Invariant

x.exists = T ⇒ x.dom >0

or, x.exists = F ⇒ x.dom = 0

Initialization

x:init {initialize the structure of x, primitive function called anywhere in PSL}

Programs

Batch_control_program

Online_control_program {not designed in this example}

PROGRAM Batch_control_program (R: request)

Overview

Accepts a request to operate on a PSL, or to install one if it does not exist.

Variables

R: request pc

{R is a local variable which holds the request}

```
Input Specification
    True  {no limitations on input}

Output Specification
    x', modified by the performance of the operation
    as follows:
    R.C  =  purge  ⇒ x' not containing u and, m="ok"
    R.C  =  add    ⇒ x' = x augmented by the unit u
                      and, m="ok"
    x.exists = T for all of the above
    if the PSL does not exist then
    x.exists = F and R.C = install ⇒ x is a PSL or
          x.exists = F and R.C ≠ install ⇒ m = message "91"

Text
    if not (x.exists)→
        if R.C = install → m := x: install (R.I)
        ▮ R.C ≠ install → m :=91
        fi
    ▮ x.exists →
        if R.C = purge → m := x: purge (R.I)
        ▮ R.C = add → m := x: add (R.I)
        ▮ ¦
        fi
    fi; m.print
END Batch_control_program

PROGRAM   Online_control_program (R:request)
    ⋮

END Online_control_program

END PSL_SYSTEM
```

MACHINE LIB:

MISSION DESCRIPTION

Functional Specification. This machine defines the type library as a collection of files. It contains, at least, the program to locate a particular file in which a unit to be purged resides.

Usage Information

Acceptance Criteria

DESIGN DOCUMENT

Defines library {the type "library"}

Overview

PROJECT in turn defines PSL which is a type defined in the highest abstract machine PSL_SYSTEM and Library defines Project.

Types

```
file: ( name: character array, data: J1)
    fun  PURGE_UNIT (n:unitname) returns m: message
         req  {this function removes units from a file}
    fun  ADD1_UNIT (n: unitname) returns m: message
         : {possibly more operations}
end file
fileunit:  (fname: character array; uname: unitname)
message: integer
J1: abstract
unitname: abstract
```

Permanent Data

L: file array

Data Invariant

All fnames contained in L are different.

Initialization

L := (0) {array initialization function}

Programs

PURGE_FILE_UNIT

ADD1_FILE_UNIT

⋮

PROGRAM PURGE_FILE_UNIT (name: fileunit) returns m: message

Overview

This program removes the file name from the pathname, and calls purge unit.

Variables

i: integer pv
{the index of the file to be purged}

m: message pv
{clear and distinct messages}

name: fileunit pc

Subprograms

file_found
!

Input Specifications

True {all input conditions allow a correct output state to be reached.}

Output Specifications

If there exists a file in L which matches name, then it is modified by PURGE_UNIT, otherwise, m = 88.

```
        Text
        i := file_found (name.fname)
        if i ≤ L.hib → m :=  L(i): PURGE_UNIT (name.uname)
        ▮ i >L.hib → m :=88
        fi

        end PURGE_FILE_UNIT

END LIB
```

MACHINE FILE

MISSION DESCRIPTION

Functional Specification

The abstract machine FILE should implement those types and operations on a PSL which pertain to files. In particular, the operations to purge units (PURGE_UNIT), add units (ADD1_UNIT), and modify the contents of a unit (CHANGE_UNIT) are implemented here. The knowledge of the contents of lower levels is hidden. That is, the actual contents of a unit are not needed for the two commands ADD1_UNIT and PURGE_UNIT. The third function was not designed. If it were, it would be necessary to create a UNIT machine.

Usage Information

This machine is invoked by performing an operation on the type file at some higher level.

Acceptance Criteria

.....

DESIGN DOCUMENTATION

Defines file

Overview

For the demonstration purposes this machine realizes the functions to add and purge a single unit of a file. The problem of "include" units is treated for the purge function.

Types

```
unit:  (uname:unitname, data:J1, info:J3)
J1: text array
text,J2: abstract

    {these types will be further
     refined when needed.  They rep-
     resent the contents of the unit's
     actual data and its unit accounting
     record.}

J3:  (count: integer, incl: unitname array ,
      stub: boolean, other: J2)
       {J3 is the unit accounting record.  For the purpose
        of this design we need

            count - the number of times a unit is
                    included in another.
            incl - an array of names of units
                   which a given unit includes.
            stub - a flag which indicates whether
                   this unit is a stub.  A stub is
                   a unit which contains no data.  It
                   serves as a placeholder for later use.}

unitname: (T1: character array
         ▮ T2: (un: character array, key: integer))
        {Since the key is optional, a unitname may consist
         of name only or name and key}

message: integer
        {a returned message number}
```

Permanent Data

f: unit array {Actually a file, but viewed at this level as an array of units}

Data Invariant

All unames are different within a single file.

A unit can be imagined as a node of a graph representing the relationship of unit inclusions. Count would then indicate the number of edges entering the node and the array incl gives the names of nodes reachable from this node.

Let f be a file. $\forall$j: f.lob $\leq$ j $\leq$ f.hib:
f(j).info.stub = true $\Rightarrow$ f(j).data.dom = 0

Other relationships between units are not of interest.

Initialization

f :=(0) {This operation (initialization) establishes the array structure}

Programs

PURGE_UNIT {Remove a unit from a file}
ADD1_UNIT {Add a single unit to a file. No include}
CHANGE_UNIT {modifies the content of a unit}

PROGRAM PURGE_UNIT (name: unitname) returns m: message

Overview

This program removes a unit from a file if it exists and is not included in another. If it satisfies these criteria, the units which are included by this unit are "unlinked" (the inclusion relationship is terminated).

Variables

i: integer pv
{an index variable}
name: unitname pc
{copying the name to a local variable so that it can be passed to another program}
m: message pv

Subprograms

unit_found {program to determine if a unit exists in the file}
delete_unit {program to remove the unit from the file}

Input Specifications

True {There are no restrictions on the input states. Therefore all input states allow the achievement of correct output state}

Output Specifications

(1) The unit exists and it is not included in another unit {count = 0}⇒
f'.dom = f.dom - 1
and the unit no longer exists
and the unit is not in any include statement
and m = 35 {good return}

(2) The unit exists and it is included in another unit
{count ≠ 0}⇒
m = 27 {error return}

(3) The unit doesn't exist ⇒
m = 13 {error return}

Performance Specifications

The performance of PURGE_UNIT for a unit which does not contain another unit is proportional to the number of units in the file divided by two (f.dom / 2). For a unit which includes other units, the performance is proportional to the number of units it includes plus one times f.dom / 2.

Text

```
i := unit_found(name); {does the unit exist}
if i ≤ f.hib cand f(i).info.count = 0 →
     delete_unit; m := 35
     {if the unit exists and is not included in
      another, delete it}
▮ i ≤ f.hib cand f(i).info.count ≠ 0 →
     m := 27
     {the unit exists and is included ... }
▮ i > f.hib → m := 13
     {the unit does not exist in the file }
fi
```

NOTES. The performance specifications can be maintained by simple sequential searches of f in unit_found and in delete_unit.

end PURGE_UNIT

SUBPROGRAM delete_unit

Overview

This program searches a unit to be deleted and adjusts the include count of units it includes. That is if the target unit has an entry in the incl array, that entry unit will be "unlinked" from the deleted unit.

Variables

```
i: integer                                  gc
   {the index of the unit to be deleted}
j,h: integer                                pv
   {the index variables}
```

Subprograms

unit_found {program to determine if a unit exists in the file}

Input Specifications

f(i) is the unit to be deleted
and f(i).info.count = 0
and f.lob ≤ i ≤ f.hib

Output Specifications

f'.dom = f.dom -1 {the unit has been deleted}
and the unit no longer exists
and ∀j such that f(i).info.incl.lob ≤ j ≤ f(i).info.incl.hib
    ∃k such that f.lob ≤ k ≤ f.hib
and f(i).info.incl(j) = f(k).uname ⇒ ∃ℓ such that f.lob ≤ ℓ ≤ f.hib
and f(i).info.incl(j) = f(ℓ).uname
and f'(k).info.count = f(k).info.count -1
    {that is, the include count for every unit
     included by this one has been decremented}

Performance Specifications

Average time proportional to f.dom/2. A linear searched algorithm is satisfactory.

Loop Invariants

~~k~~ f(i).info.incl.lob $\leq$ k < j

f'(ℓ).info.count = f(ℓ).info.count - 1

{for all units which have been looked at,
the count has been decremented}

Text

```
j := f(i).info.incl.lob;
     {look at every unit in the include array}
do j ≤ f(i).info.incl.hib ↪
     h := unit_found (f(i).info.incl(j));
     f(h).info.count := f(h).info.count - 1;
     j := j + 1
od;
     {At this point, all include counts have been reset.
      Now remove the unit. }
     f:swap(i,f.hib); f:hirem
     {The unit has been deleted.}
```

Notes

The operations swap and hirem can be efficiently implemented with pointers.

The performance specifications will be satisfied if unit_found uses a linear search algorithm.

end delect_unit

SUBPROGRAM unit_found (n: unitname) returns i: integer

Overview

This program searches a file (array of units) for a particular unit. If it is located, an index in the file array is returned. Otherwise, a value which is not a valid index is returned.

Variables

i: integer
{the index of a unit in the file}
n: unitname

Input Specifications

True {No restrictions on the input states}

Output Specifications

{Either the file has been searched and the unit located, or the file does not contain the unit}

$\forall k$: f.lob $\leq$ k < i
$\Rightarrow$ [f(k).uname = n
and (f(i).uname = n or i > f.hib)
or ((f(i).uname.key = n and f(i).uname.un = n) or i > f.hib)]

Performance Specifications

Average time requirements are satisfied by a linear search algorithm.

Loop Invariants

$\forall k$: f.lob $\leq$ k < i $\Rightarrow$ f(k).uname $\neq$ n

$\forall k$: f.lob $\leq$ k < i (f(k).uname.key $\neq$ n or f(k).uname.un $\neq$ n)

{We haven't found it, or we are still looking, or it doesn't exist}

```
        Text
        i := f.lob;
        if  n.tag = T1 →
                {if no key was provided}
            do i ≤ f.hib cand f(i).uname ≠ n →
                i := i + 1
            od
        ▮  n.tag = T2 →
                {if a key was provided}
            do i ≤ f.hib cand ((f(i).uname.un ≠ n.un
                                cor f(i).uname.key ≠ n.key)→
                i := i + 1
            od
        fi
    end unit_found
```

```
PROGRAM  ADD1_UNIT(AU:  (aname: unitname,
         data: J1)) returns  (m: message)
```

Overview

This program will add a single unit which contains no includes. The operation is accomplished by constructing an intermediate data x. Clear and distinct messages are transmitted for each different operation.

Variables

x: unit	pv
i: integer	pv
AU: (aname: unitname, data: J1)	pc
m: message	pv

Subprograms

unit_found

Input Specification

There are no INCLUDE statements in the unit to be added. One add request at a time.

Output Specification

(1) (∀ i:f.lob≤i≤f.hib: f(i).uname≠AU.aname or f.dom=0)
and AU.data.dom ≠ 0
f'.dom = f.dom + 1 and
∃j f'(j).uname = AU.aname
and f'(j).data = AU.data
and f'(j).info.count = 0
and f'(j).info.incl.dom = 0
and f'(j).info.stub = FALSE
and m=16

(2) ∀i: f.lob≤i≤f.hib: f(i).uname≠AU.uname
or f.dom = 0 and AU.data.dom = 0
f'.dom = f.dom + 1 and
∃j f'(j).uname = AU.aname
and f'(j).data = AU.data {empty}
and f'(j).info.count = 0
and f'(j).info.incl.dom = 0
and f'(j).info.stub = TRUE
and m=38

(3) ∃i: f.lob≤i≤f.hib: f(i).uname=AU.aname and
f(i).info.stub = TRUE
f'(i).info.stub = FALSE
and f'(i).data = AU.data
and m=41

(4) ∃i: f.lob≤i≤f.hib: f(i).uname = AU.aname and
f(i).info.stub = FALSE
and m=47

Performance Specification

Average time proportional to f.dom/2. A linear search algorithm for unit_found satisfies the above requirements.

```
Text
unit_found (n:AU.aname);
if i > f.hib and AU.data.dom ≠ 0 →
    {create a new unit, not a stub}
    x.uname := AU.aname;
    x.data := AU.data;
    x.info.count := 0;
    x.info.incl := (0);
    x.info.stub := F;
    f:hiext(x); m := 16
▮ i ≤ f.hib and AU.data.dom ≠ 0
    and f(i).info.stub = T →
    {change a stub in a data unit}
    f(i).data := AU.data;
    f(i).info.stub := F; m := 41
▮ i >hib and AU.data.dom ≠ 0 →
    {create a new stub}
    x.uname := AU.aname;
    x.data := (0);
    x.info.count := 0;
    x.info.incl := (0);
    x.info.stub := T;
    f:hiext(x); m := 38
▮ i ≤ hib and f(i).info.stub = F and
    AU.data.dom = 0 →
    {the attempt to make a stub out of an
     existing unit is refused}
    m := 47
fi

end ADD1_UNIT

END FILE
```

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.